FPGAs: Instant Access

Author(s): Clive Max Maxfield
ISBN: 978-0-7506-8974-8

Table of Contents

About the Author, Page ix

Chapter 1 - The Fundamentals, Pages 1-12

Chapter 2 - FPGA Architectures, Pages 13-48

Chapter 3 - Programming (Configuring) an FPGA, Pages 49-60

Chapter 4 - FPGA vs. ASIC Designs, Pages 61-73

Chapter 5 - “Traditional” Design Flows, Pages 75-106

Chapter 6 - Other Design Flows, Pages 107-153

Chapter 7 - Using Design Tools, Pages 155-184

Chapter 8 - Choosing the Right Device, Pages 185-196

Index, Pages 197-204

VISIT…

About the Author

Clive "Max" Maxfield is six feet tall, outrageously handsome, English and proud of it. In addition to being a hero, trendsetter, and leader of fashion, he is widely regarded as an expert in all aspects of electronics and computing (at least by his mother).

After receiving his B.Sc. in Control Engineering in 1980 from Sheffield Polytechnic (now Sheffield Hallam University), England, Max began his career as a designer of central processing units for mainframe computers. During his career, he has designed everything from ASICs to PCBs and has meandered his way through most aspects of Electronics Design Automation (EDA). To cut a long story short, Max now finds himself President of TechBites Interactive (**www.techbites.com**). A marketing consultancy, TechBites specializes in communicating the value of its clients' technical products and services to non-technical audiences through a variety of media, including websites, advertising, technical documents, brochures, collaterals, books, and multimedia.

In addition to numerous technical articles and papers appearing in magazines and at conferences around the world, Max is also the author and co-author of a number of books, including *Bebop to the Boolean Boogie (An Unconventional Guide to Electronics)*, *Designus Maximus Unleashed (Banned in Alabama)*, *Bebop BYTES Back (An Unconventional Guide to Computers)*, *EDA: Where Electronics Begins*, *The Design Warrior's Guide to FPGAs*, and *How Computers Do Math* (**www.diycalculator.com**).

In his spare time (Ha!), Max is co-editor and co-publisher of the web-delivered electronics and computing hobbyist magazine *EPE Online* (**www.epemag.com**). Max also acts as editor for the Programmable Logic DesignLine website (**www.pldesignline.com**) and for the iDESIGN section of the Chip Design Magazine website (**www.chipdesignmag.com**).

On the off-chance that you're still not impressed, Max was once referred to as an *"industry notable"* and a *"semiconductor design expert"* by someone famous who wasn't prompted, coerced, or remunerated in any way!

Chapter 1

The Fundamentals

In an Instant
Why Use FPGAs?
Applications
Some Technology Background
Fusible-link Technology
FPGA Programming Technologies
Instant Summary

FPGA Definitions

- *Field programmable gate arrays* (FPGAs) are digital integrated circuits (ICs) that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks. Design engineers can configure, or program, such devices to perform a tremendous variety of tasks.
- Depending on how they are implemented, some FPGAs may only be programmed a single time, while others may be reprogrammed over and over again. Not surprisingly, a device that can be programmed only one time is referred to as *one-time programmable* (OTP).
- The "field programmable" portion of the FPGA's name refers to the fact that its programming takes place "in the field" (as opposed to devices whose internal functionality is hardwired by the manufacturer). This may mean that FPGAs are configured in the laboratory, or it may refer to modifying the function of a device resident in an electronic system that has already been deployed in the outside world. If a device is capable of being programmed while remaining resident in a higher-level system, it is referred to as being *in-system programmable* (ISP).
- In this book, we'll be referring to programmable logic devices (PLDs), application-specific integrated circuits (ASICs), application-specific standard parts (ASSPs), and—of course—FPGAs.

WHY USE FPGAs?

Various aspects of PLDs, ASICs, and FPGAs will be discussed later in more detail. For now, we need only be aware that PLDs are devices whose internal

architecture is predetermined by the manufacturer, but are created in such a way that they can be configured by engineers in the field to perform a variety of different functions. In comparison to an FPGA, however, these devices contain a relatively limited number of logic gates, and the functions they can be used to implement are much smaller and simpler.

At the other end of the spectrum are ASICs and ASSPs that can contain hundreds of millions of logic gates and can be used to create incredibly large and complex functions. ASICs and ASSPs are based on the same design processes and manufacturing technologies. Both are custom-designed to address a specific application, the only difference being that an ASIC is designed and built to order for use by a specific company, while an ASSP is marketed to multiple customers.

ALERT!

When we use the term ASIC from now on, it may be assumed that we are also referring to ASSPs unless otherwise noted or where such interpretation is inconsistent with the context.

Although ASICs offer the ultimate in size (number of transistors), complexity, and performance, designing and building one is an extremely time-consuming and expensive process, with the added disadvantage that the final design is "frozen in silicon" and cannot be modified without creating a new version of the device.

Thus, FPGAs occupy a middle ground between PLDs and ASICs because their functionality can be customized in the field like PLDs, but they can contain millions of logic gates and be used to implement extremely large and complex functions that previously could be realized using only ASICs.

—Technology Trade-offs—

- The cost of an FPGA design is much lower than that of an ASIC (although the ensuing ASIC components are much cheaper in large production runs).
- Implementing design changes is much easier in FPGAs.
- Time to market for FPGAs is much faster.

FPGAs make many small, innovative design companies viable because—in addition to their use by large system design houses—FPGAs facilitate "Fred-in-the-shed"-type operations. This means they allow individual engineers or small groups of engineers to realize their hardware and software concepts on an FPGA-based test platform without having to incur the enormous nonrecurring engineering (NRE) costs or purchase the expensive toolsets associated with ASIC designs. Hence, there were estimated to be only 1,500 to 4,000 ASIC design starts and 5,000 ASSP design starts in 2003 (these numbers are falling dramatically year by year), as opposed to an educated "guesstimate" of around 450,000 FPGA design starts in the same year.

Insider Info

These design-start numbers are a little hard to pin down because it's difficult to get everyone to agree what a "design start" actually is. In the case of an ASIC, for example, should we include designs that are canceled in the middle, or should we only consider designs that make it all the way to tape-out? Things become even fluffier when it comes to FPGAs due to their reconfigurability. Perhaps even more telling is the fact that, after pointing me toward an FPGA-centric industry analyst's Web site, a representative from one FPGA vendor added, "But the values given there aren't very accurate." When I asked why, he replied with a sly grin, "Mainly because we don't provide him with very good data!"

APPLICATIONS

When they first arrived on the scene in the mid-1980s, FPGAs were largely used to implement glue logic, medium-complexity state machines, and relatively limited data processing tasks. During the early 1990s, as the size and sophistication of FPGAs started to increase, their big markets at that time were in the telecommunications and networking arenas, both of which involved processing large blocks of data and pushing that data around. Later, toward the end of the 1990s, the use of FPGAs in consumer, automotive, and industrial applications underwent a humongous growth spurt.

FPGAs are often used to prototype ASIC designs or to provide a hardware platform on which to verify the physical implementation of new algorithms. However, their low development cost and short time-to-market mean that they are increasingly finding their way into final products (some of the major FPGA vendors actually have devices they specifically market as competing directly against ASICs).

High-performance FPGAs containing millions of gates are currently available. Some of these devices feature embedded microprocessor cores, high-speed input/output (I/O) devices, and the like. The result is that today's FPGAs can be used to implement just about anything, including communications devices and software-defined radio; radar, image, and other digital signal processing (DSP) applications; and all the way up to *system-on-chip* (SoC) components that contain both hardware and software elements.

FAQs

What are the major market segments for FPGAs?

- *ASIC and custom silicon*: FPGAs are increasingly being used to implement designs that previously were realized by using only ASICs and custom silicon.
- *Digital signal processing*: Today's FPGAs can contain embedded multipliers, dedicated arithmetic routing, and large amounts of on-chip RAM, all of which facilitate DSP operations. When coupled with the massive parallelism provided

by FPGAs, this results in outperforming the fastest DSP chips by a factor of 500 or more.

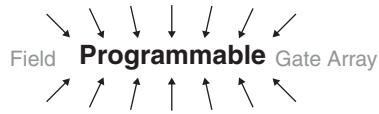
- *Embedded microcontrollers*: Low-cost microcontrollers, which contain on-chip program and instruction memories, timers and I/O peripherals wrapped around a processor core, are used in small control functions. With falling FPGA prices, however, and increased capability to implement a soft processor core combined with a selection of custom I/O functions, FPGAs are becoming increasingly attractive for embedded control applications.
- *Physical layer communications*: FPGAs have long been used for the glue logic that interfaces between physical layer communication chips and high-level networking protocol layers. Now high-end FPGAs can contain multiple high-speed transceivers, which means that communications and networking functions can be consolidated into a single device.
- *Reconfigurable computing (RC)*: FPGAs have created this new market segment. This refers to exploiting the inherent parallelism and reconfigurability provided by FPGAs to "hardware accelerate" software algorithms. Various companies are currently building huge FPGA-based reconfigurable computing engines for tasks ranging from hardware simulation to cryptography analysis to discovering new drugs.

SOME TECHNOLOGY BACKGROUND

The first FPGA devices contained only a few thousand simple logic gates (or the equivalent), and the flows used to design these components—predominantly based on the use of schematic capture—were easy to understand and use. By comparison, today's FPGAs are incredibly complex, and there are more design tools, flows, and techniques than you can swing a stick at. In this section we'll look at some technology basics.

Key Concept

What distinguishes an FPGA from an ASIC is embodied in the name:



Fusible-link Technology

Let's first consider a very simple programmable function with two inputs called *a* and *b* and a single output *y* (Figure 1-1).

The inverting NOT gates associated with the inputs mean that each input is available in both its *true* (unmodified) and *complemented* (inverted) form. Observe the locations of the potential links. In the absence of any of these

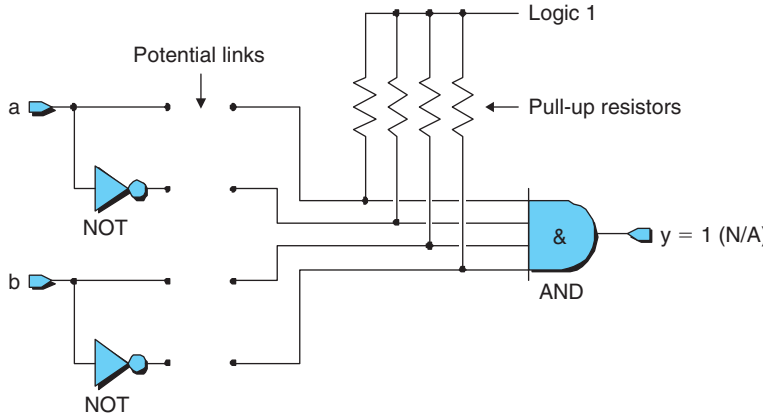


FIGURE 1-1 A simple programmable function.

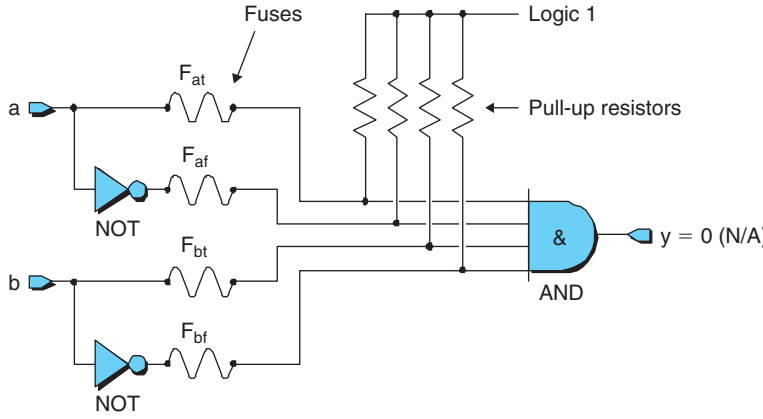


FIGURE 1-2 Augmenting the device with unprogrammed fusible links.

links, all of the inputs to the AND gate are connected via pull-up resistors to a logic 1 value. In turn, this means that the output *y* will always be driving a logic 1, which makes this circuit a very boring one in its current state. To make this function more interesting, we need a mechanism that allows us to establish one or more of the potential links. This mechanism is *fusible-link technology.* In this case, the device is manufactured with all of the links in place, with each link referred to as a *fuse* (Figure 1-2).

These fuses are similar to the fuses you find in household products like a television. If anything untoward occurs such that the television starts to consume too much power, its fuse will burn out, resulting in an open circuit, which protects the rest of the unit from harm. Of course, the fuses in silicon chips are formed using the same processes that are employed to create the transistors and wires on the chip, so they're microscopically small.

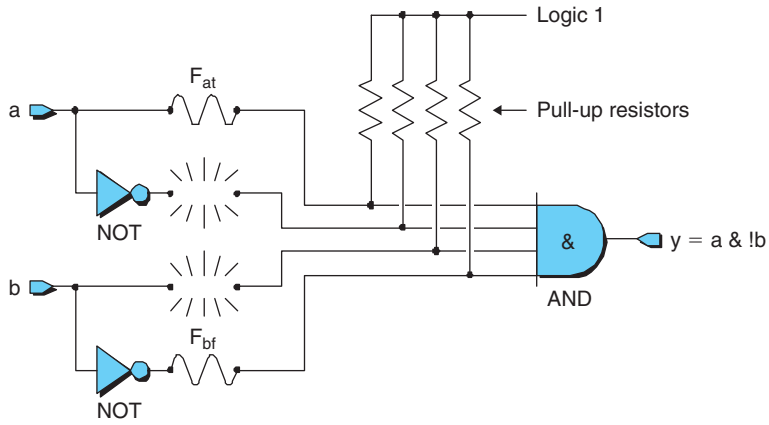


FIGURE 1-3 Programmed fusible links.

Although fusible-link technology is not used in today's FPGAs, it sets the stage for understanding technologies that are, so we'll explore it briefly. When you purchase a programmable device based on fusible links, all the fuses are initially intact. This means that, in its unprogrammed state, the output from our example function is always logic 0. (Any 0 presented to the input of an AND gate will cause its output to be 0, so if input *a* is 0, the output from the AND will be 0. Alternatively, if input *a* is 1, then the output from its NOT gate—which we shall call *!a*—will be 0, and once again the output from the AND will be 0. A similar situation occurs in the case of input *b*.)

The point is that design engineers can selectively remove undesired fuses by applying pulses of relatively high voltage and current to the device's inputs. For example, consider what happens if we remove fuses F_{af} and F_{bt} (Figure 1-3).

Removing these fuses disconnects the complementary version of input *a* and the true version of input *b* from the AND gate (the pull-up resistors associated with these signals cause their associated inputs to the AND to be presented with logic 1 values). This leaves the device to perform its new function, which is *y = a & !b*. (The "&" character in this equation is used to represent the AND, while the "!" character is used to represent the NOT.) This process of removing fuses is typically called *programming* the device, but it may also be called *blowing* the fuses or *burning* the device.

Key Concept

Devices based on fusible-link technologies are *one-time programmable*, or OTP, because once a fuse has been blown, it can't be replaced and there's no going back.

FPGA Programming Technologies

Three different major technologies are in use today for programming FPGAs: antifuse, SRAM, and FLASH EPROM.

Antifuse Technology

As a diametric alternative to fusible-link technologies, we have their antifuse counterparts, in which each configurable path has an associated link called an antifuse. In its unprogrammed state, an antifuse has such a high resistance that it may be considered an open circuit (a break in the wire).

How It Works

Figure 1-4 shows how the device appears when first purchased. However, antifuses can be selectively "grown" (programmed) by applying pulses of relatively high voltage and current to the device's inputs. For example, if we add the antifuses associated with the complementary version of input a and the true version of input b, our device will now perform the function y = !a & b (Figure 1-5).

An antifuse commences life as a microscopic column of amorphous (noncrystalline) silicon linking two metal tracks. In its unprogrammed state, the amorphous silicon acts as an insulator with a very high resistance in excess of 1 billion ohms (Figure 1-6a).

The act of programming this particular element effectively "grows" a link, known as a via, by converting the insulating amorphous silicon in conducting polysilicon (Figure 1-6b).

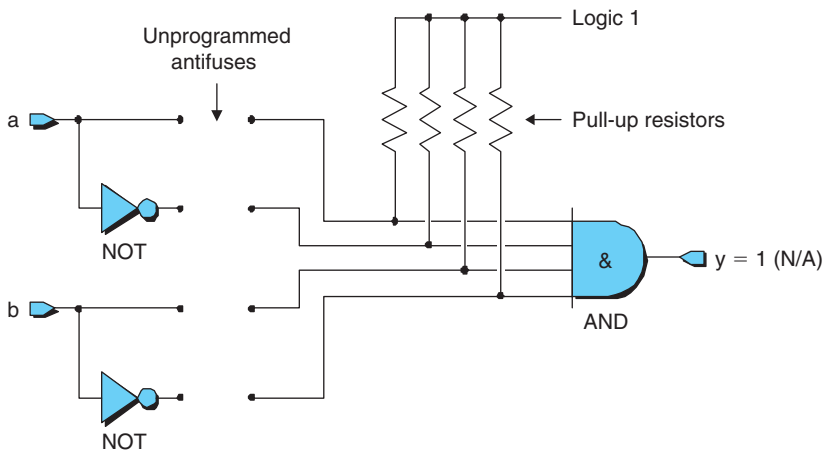


FIGURE 1-4 Unprogrammed antifuse links.

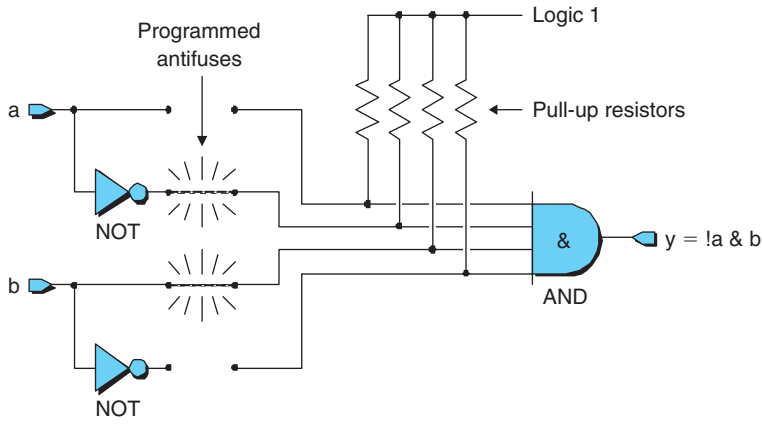


FIGURE 1-5 Programmed antifuse links.

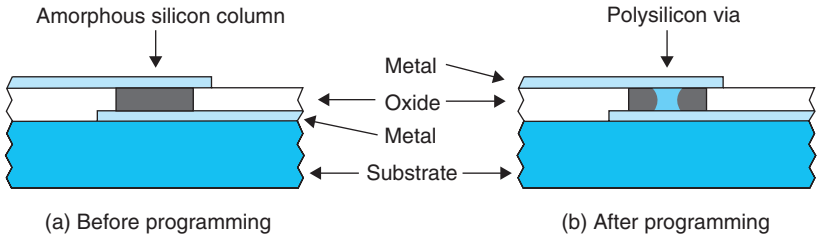


FIGURE 1-6 Growing an antifuse.

—Technology Trade-offs—

- Not surprisingly, devices based on antifuse technologies are OTP, because once an antifuse has been grown, it cannot be removed, and there's no changing your mind.
- Antifuse devices tend to be faster and require lower power.

SRAM-based Technology

There are two main versions of semiconductor RAM devices: dynamic RAM (DRAM) and static RAM (SRAM). DRAM technology is of very little interest with regard to programmable logic, so we will focus on SRAM.

Key Concept

SRAM is currently the dominant FPGA technology.

The "static" qualifier associated with SRAM means that—once a value has been loaded into an SRAM cell—it will remain unchanged unless it is specifically altered or until power is removed from the system.

How It Works

Consider the symbol for an SRAM-based programmable cell (Figure 1-7).

The entire cell comprises a multitransistor SRAM storage element whose output drives an additional control transistor. Depending on the contents of the storage element (logic 0 or logic 1), the control transistor will be either OFF (disabled) or ON (enabled).

SRAM is currently the dominant FPGA technology.

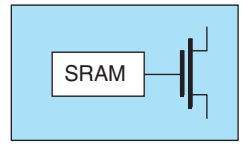


FIGURE 1-7 An SRAM-based programmable cell.

—Technology Trade-offs—

- A disadvantage of SRAM-based programmable devices is that each cell consumes a significant amount of silicon real estate because the cells are formed from four or six transistors configured as a latch.
- Another disadvantage is that the device's configuration data (programmed state) will be lost when power is removed from the system, so these devices always have to be reprogrammed when the system is powered on.
- Advantages are that such devices can be reprogrammed quickly and easily, and SRAM uses a standard fabrication technology that is always being improved upon.

FLASH-based Technologies

A relatively new technology known as FLASH is being used in some FPGAs today. This technology grew out of an earlier technology known as *erasable programmable read-only memory* (EPROM) that allows devices to be programmed, erased, and reprogrammed with new data. We will first look at how EPROMs work before discussing FLASH.

An EPROM transistor has the same basic structure as a standard MOS transistor, but with the addition of a second polysilicon floating gate isolated by layers of oxide (Figure 1-8).

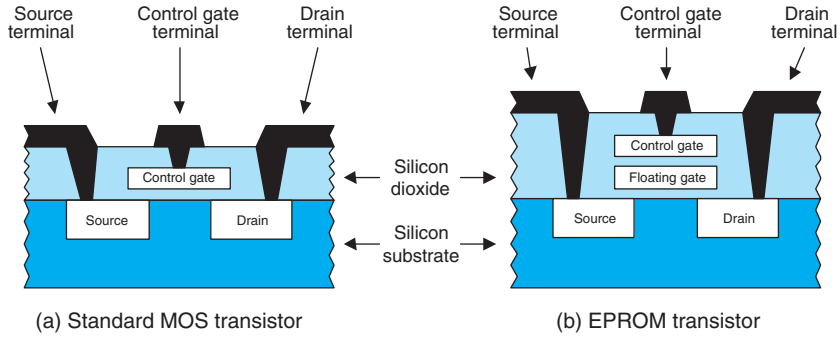


FIGURE 1-8 Standard MOS versus EPROM transistors.

How It Works

In its unprogrammed state, the floating gate is uncharged and doesn't affect the normal operation of the control gate. In order to program the transistor, a relatively high voltage (on the order of 12V) is applied between the control gate and drain terminals. This causes the transistor to be turned hard on, and energetic electrons force their way through the oxide into the floating gate in a process known as hot (high energy) electron injection. When the programming signal is removed, a negative charge remains on the floating gate. This charge is very stable and will not dissipate for more than a decade under normal operating conditions. The stored charge on the floating gate inhibits the normal operation of the control gate and, thus, distinguishes those cells that have been programmed from those that have not. This means we can use such a transistor to form a memory cell (Figure 1-9).

In its unprogrammed state, as provided by the manufacturer, all of the floating gates in the EPROM transistors are uncharged. In this case, placing a row line in its active state will turn on all of the transistors connected to that row, thereby causing all of the column lines to be pulled down to logic 0 via their respective transistors. In order to program the device, engineers can use the inputs to the device to charge the floating

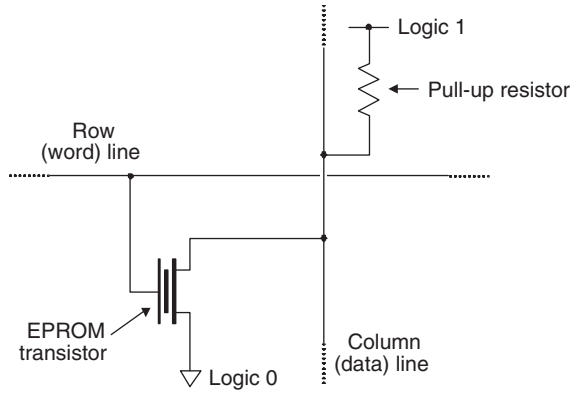


FIGURE 1-9 An EPROM transistor-based memory cell.

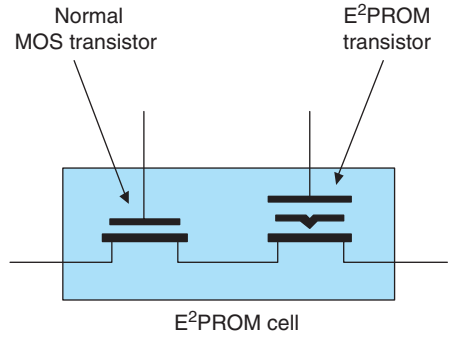


FIGURE 1-10 An E2PROM—cell.

gates associated with selected transistors, thereby disabling those transistors. In these cases, the cells will appear to contain logic 1 values.

These devices were initially intended for use as programmable memories, but the same technology was applied to more general-purpose PLDs, which became known as erasable PLDs (EPLDs). The main problems with EPROMs are their expensive packages (with quartz windows through which ultraviolet (UV) radiation is used to erase the device) and the time it takes to erase them, on the order of 20 minutes.

The next rung up the technology ladder was electrically erasable programmable read-only memories (EEPROMs or E²PROMs). An E²PROM cell is approximately 2.5 times larger than an equivalent EPROM cell because it comprises two transistors and the space between them (Figure 1-10).

The E2PROM transistor is similar to an EPROM transistor in that it contains a floating gate, but the insulating oxide layers surrounding this gate are very much thinner. The second transistor can be used to erase the cell electrically. E2PROMs first saw the light of day as computer memories, but the same technology was eventually applied to PLDs, which became known as electrically erasable PLDs (EEPLDs or E2PLDs).

FLASH can trace its ancestry to both EPROM and EEPROM technologies. The name "FLASH" was originally coined to reflect this technology's rapid erasure times compared to EPROM. Components based on FLASH can employ a variety of architectures. Some have a single floating gate transistor cell with the same area as an EPROM cell, but with the thinner oxide layers characteristic of an E2PROM component. These devices can be electrically erased, but only by clearing the whole device or large portions thereof. Other architectures feature a two-transistor cell similar to that of an E2PROM cell, thereby allowing them to be erased and reprogrammed on a word-by-word basis.

—Technology Trade-offs—

- FLASH FPGAs are nonvolatile like antifuse FPGAs, but they are also reprogrammable like SRAM FPGAs.
- FLASH FPGAs use a standard fabrication process like SRAM FPGAs and use lower power like antifuse FPGAs.
- FLASH FPGAs are relatively fast.

INSTANT SUMMARY

TABLE 1-1 Summary of Programming Technologies

Technology	Symbol	Predominantly associated with ...
Fusible-link		SPLDs
Antifuse		FPGAs
EPROM		SPLDs and CPLDs
E^2PROM/FLASH		SPLDs and CPLDs (some FPGAs)
SRAM		FPGAs (some CPLDs)

Chapter 2

FPGA Architectures

In an Instant

More on Programming Technologies
- SRAM-based Devices
- Antifuse-based Devices
- E^2PROM/FLASH-based devices
- Hybrid FLASH-SRAM Devices

Fine-, Medium-, and Coarse-grained Architectures

Logic Blocks
- MUX-based
- LUT-based
- LUT versus Distributed RAM versus SR

CLBs versus LABs versus Slices
- Logic Cells/Logic Elements
- Slicing and Dicing
- CLBs and LABs
- Distributed RAMs and Shift Registers

Embedded RAMs

Embedded Multipliers, Adders, etc.

Embedded Processor Cores
- Hard Microprocessor Cores
- Soft Microprocessor Cores

Clock Managers
- Clock Trees
- Clock Managers

General-purpose I/O
- Configurable I/O Standards
- Configurable I/O Impedances
- Core versus I/O Supply Voltages

Gigabit Transceivers
- Multiple Standards

Intellectual Property (IP)
- Handcrafted IP
- IP Core Generators

System Gates versus Real Gates

Instant Summary

Definitions

In this chapter we'll discuss a plethora of architectural features of FPGAs. But first, some definitions.

- The term *fabric* is used throughout this book. In the context of a silicon chip, this refers to the underlying structure of the device, sort of like the phrase "the underlying fabric of civilization."
- When we talk about the *geometry* of an IC, we are referring to the size of the individual structures constructed on the chip, such as the portion of a field-effect transistor (FET) known as its *channel*. These structures are incredibly small. In the early to mid-1980s, devices were based on 3 μm geometries, which means

that their smallest structures were 3 millionths of a meter in size. Now, devices at 0.09 μm have appeared.

- Any geometry smaller than around 0.5 μm is referred to as *deep submicron (DSM)*. At some point that is not well defined (or that has multiple definitions depending on whom one is talking to), we move into the *ultradeep submicron (UDSM)* realm.
- We'll also discuss the important topic of *intellectual property* (IP) in this chapter. This term refers to functional design blocks that have already been developed and can be purchased to put into an IC design. IP blocks can range all the way up to sophisticated communications functions and microprocessors. The more complex functions, like microprocessors, may be referred to as "cores."

Insider Info

When geometries dropped below 1 μm, things became a little awkward, not the least because it's a pain to keep saying things like "zero point one three microns." For this reason, when conversing it's becoming common to talk in terms of nano, where one nano (short for nanometer) equates to a thousandth of a micron. Instead of mumbling "point zero nine microns" (0.09 μm), one can simply proclaim "ninety nano" (90 nano) and have done with it.

MORE ON PROGRAMMING TECHNOLOGIES

SRAM-based Devices

As seen in Chapter 1, the majority of FPGAs are based on the use of SRAM configuration cells, which means that they can be configured over and over again. The main advantages of this programming technology are that new design ideas can be quickly implemented and tested, while evolving standards and protocols can be accommodated relatively easily. Furthermore, when the system is first powered up, the FPGA can initially be programmed to perform one function such as a self-test or board/system test, and it can then be reprogrammed to perform its main task.

Another big advantage of the SRAM-based approach is that these devices are at the forefront of technology. FPGA vendors can leverage the fact that many other companies specializing in memory devices expend tremendous resources on *research and development (R&D)* in this area. Furthermore, the SRAM cells are created using exactly the same CMOS technologies as the rest of the device, so no special processing steps are required in order to create these components.

Unfortunately, there's no such thing as a free lunch. One downside of SRAM-based devices is that they have to be reconfigured every time the system is powered up. This either requires the use of a special external memory device (which has an associated cost and consumes real estate on the board) or of an on-board microprocessor (or some variation of these techniques—see also Chapter 3).

Security Issues

Another consideration with regard to SRAM-based devices is that it can be difficult to protect your *intellectual property*, or IP, in the form of your design. This is because the configuration file used to program the device is stored in some form of external memory.

ALERT!

Remember that there are reverse-engineering companies all over the world specializing in the recovery of "design IP." And there are also a number of countries whose governments turn a blind eye to IP theft so long as the money keeps rolling in (you know who you are). So if a design is a high-profit item, you can bet that there are folks out there who are ready and eager to replicate it while you're not looking.

In reality, the real issue here is not related to those stealing your IP by reverse-engineering the contents of the configuration file, but rather their ability to clone your design, irrespective of whether they understand how it works. Using readily available technology, it is relatively easy for someone to take a circuit board, put it on a "bed of nails" tester, and quickly extract a complete netlist for the board. This netlist can subsequently be used to reproduce the board. Now the only task remaining for the nefarious scoundrels is to copy your FPGA configuration file from its boot PROM (or EPROM, E2PROM, or whatever), and they have a duplicate of the entire design.

On the bright side, some of today's SRAM-based FPGAs support the concept of *bitstream encryption*. In this case, the final configuration data is encrypted before being stored in the external memory device. The encryption key itself is loaded into a special SRAM-based register in the FPGA via its JTAG port (see also Chapter 3). In conjunction with some associated logic, this key allows the incoming encrypted configuration bitstream to be decrypted as it's being loaded into the device.

The command/process of loading an encrypted bitstream automatically disables the FPGA's read-back capability. This means that you will typically use unencrypted configuration data during development (where you need to use read-back) and then start to use encrypted data when you move into production. (You can load an unencrypted bitstream at any time, so you can easily load a test configuration and then reload the encrypted version.)

—Technology Trade-offs—

- The main downside to the encrypted bitstream scheme is that you require a battery backup on the circuit board to maintain the contents of the encryption key register in the FPGA when power is removed from the system. This battery will have a lifetime of years or decades because it need only maintain a single register in the device, but it does add to the size, weight, complexity, and cost of the board.

Antifuse-based Devices

Unlike SRAM-based devices, which are programmed while resident in the system, antifuse-based devices are programmed off-line using a special device programmer. The proponents of antifuse-based FPGAs are proud to point to an assortment of (not-insignificant) advantages:

1. First, these devices are **nonvolatile** (their configuration data remains when the system is powered down), which means that they are immediately available as soon as power is applied to the system, and they don't require an external memory chip to store their configuration data, which saves the cost of an additional component and saves real estate on the board.
2. Another noteworthy advantage of antifuse-based FPGAs is the fact that their interconnect structure is naturally **rad hard**, which means they are relatively immune to the effects of radiation. This is of particular interest in the case of military and aerospace applications because the state of a configuration cell in an SRAM-based component can be "flipped" if that cell is hit by radiation (of which there is a lot in space). By comparison, once an antifuse has been programmed, it cannot be altered in this way.

ALERT!

It should be noted that any flip-flops in these devices remain sensitive to radiation, so chips intended for radiation-intensive environments must have their flip-flops protected by triple redundancy design. This refers to having three copies of each register and taking a majority vote (ideally all three registers will contain identical values, but if one has been "flipped" such that two registers say 0 and the third says 1, then the 0s have it, or vice versa if two registers say 1 and the third says 0).

3. Perhaps the most significant advantage of antifuse-based FPGAs is that their **configuration data is buried deep inside them, making it almost impossible to reverse-engineer the design**. By default, it is possible for the device programmer to read this data out because this is actually how the programmer works. As each antifuse is being processed, the device programmer keeps on testing it to determine when that element has been fully programmed; then it moves on to the next antifuse. Furthermore, the device programmer can be used to automatically verify that the configuration was performed successfully (this is well worth doing when you're talking about devices containing 50 million plus programmable elements). Once the device has been programmed, however, it is possible to set (grow) a special security antifuse that subsequently prevents any programming data (in the form of the presence or absence of antifuses) from being read out of the device. Even if the device is decapped (its top is removed), programmed and unprogrammed antifuses appear to be identical, and the fact that all of

the antifuses are buried in the internal metallization layers makes reverse-engineering close to impossible.

Of course, the main disadvantage associated with antifuse-based devices is that they are OTP, so once you've programmed one, its function is set in stone. This makes these components a poor choice for use in a development or prototyping environment.

—Technology Trade-offs—

- Vendors of antifuse-based FPGAs may tout the fact that an antifuse-based device consumes only 20 percent (approximately) of the standby power of an equivalent SRAM-based component, that their operational power consumption is also significantly lower, and that their interconnect-related delays are smaller. Also, they might casually mention that an antifuse is much smaller and thus occupies much less real estate on the chip than an equivalent SRAM cell.
- They may neglect to mention, however, that antifuse devices also require extra programming circuitry, including a large, hairy programming transistor for each antifuse.
- Also, antifuse technology requires the use of around three additional process steps after the main manufacturing process has been qualified. For this (and related) reason, antifuse devices are always at least one—and usually several—generations (technology nodes) behind SRAM-based components, which effectively wipes out any speed or power consumption advantages that might otherwise be of interest.

E^2PROM/FLASH-based Devices

E^2PROM- or FLASH-based FPGAs are similar to their SRAM counterparts in that their configuration cells are connected together in a long shift-register-style chain. These devices can be configured off-line using a device programmer. Alternatively, some versions are in-system programmable, or ISP, but their programming time is about three times that of an SRAM-based component. However, they do have some advantages:

1. Once programmed, the data they contain is **nonvolatile**, so these devices would be "instant on" when power is first applied to the system.
2. With regard to protection, some of these devices use the concept of a **multibit key**, which can range from around 50 bits to several hundred bits in size. Once you've programmed the device, you can load your user-defined key (bit-pattern) to secure its configuration data. After the key has been loaded, the only way to read data out of the device, or to write new data into it, is to load a copy of your key via the JTAG port (this port is discussed later in this chapter and in Chapter 3). The fact that the JTAG port in today's devices runs at around 20 MHz means that it would take billions of years to crack the key by exhaustively trying every possible value.

3. Two-transistor E^2PROM and FLASH cells are approximately 2.5 times the size of their one-transistor EPROM cousins, but they are still way **smaller than their SRAM counterparts**. This means that the rest of the logic can be much closer together, thereby **reducing interconnect delays**.

On the downside, these devices require around five additional process steps on top of standard CMOS technology, which results in their lagging behind SRAM-based devices by one or more generations (technology nodes). Last but not least, these devices tend to have relatively high static power consumption due to their containing vast numbers of internal pull-up resistors.

Hybrid FLASH-SRAM Devices

Last but not least, there's always someone who wants to add yet one more ingredient to the cooking pot. In the case of FPGAs, some vendors offer esoteric combinations of programming technologies. For example, consider a device where each configuration element is formed from the combination of a FLASH (or E^2PROM) cell and an associated SRAM cell.

In this case, the FLASH elements can be preprogrammed. Then, when the system is powered up, the contents of the FLASH cells are copied in a massively parallel fashion into their corresponding SRAM cells. This technique gives you the nonvolatility associated with antifuse devices, which means the device is immediately available when power is first applied to the system. But unlike an antifuse-based component, you can subsequently use the SRAM cells to reconfigure the device while it remains resident in the system. Alternatively, you can reconfigure the device using its FLASH cells either while it remains in the system or off-line by means of a device programmer.

FINE-, MEDIUM-, AND COARSE-GRAINED ARCHITECTURES

It is common to categorize FPGA offerings as being either fine grained or coarse grained. In order to understand what this means, we first need to remind ourselves that the main feature that distinguishes FPGAs from other devices is that their underlying fabric predominantly consists of large numbers of relatively simple programmable logic block "islands" embedded in a "sea" of programmable interconnect (Figure 2-1).

In the case of a *fine-grained architecture*, each logic block can be used to implement only a very simple function. For example, it might be possible to configure the block to act as any 3-input function, such as a primitive logic gate (AND, OR, NAND, etc.) or a storage element (D-type flip-flop, D-type latch, etc.).

In the case of a *coarse-grained architecture*, each logic block contains a relatively large amount of logic compared to their fine-grained counterparts. For example, a logic block might contain four 4-input LUTs, four multiplexers, four D-type flip-flops, and some fast carry logic (see the following topics in this chapter for more details).

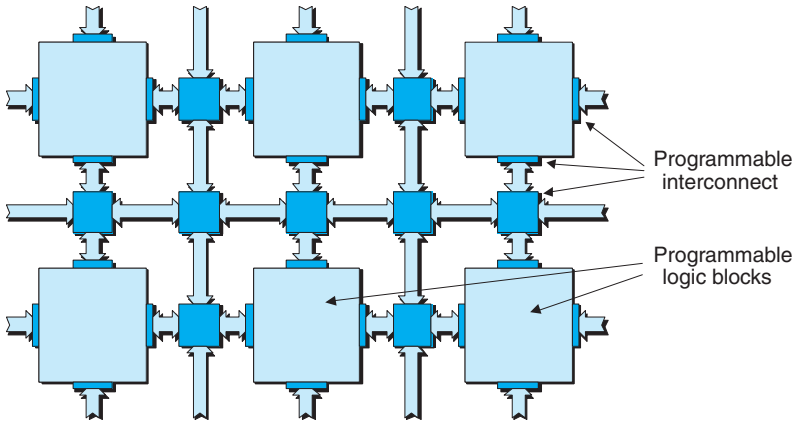


FIGURE 2-1 Underlying FPGA fabric.

An important consideration with regard to architectural granularity is that fine-grained implementations require a relatively large number of connections into and out of each block compared to the amount of functionality that can be supported by those blocks. As the granularity of the blocks increases to *medium-grained* and higher, the amount of connections into the blocks decreases compared to the amount of functionality they can support. This is important because the programmable interblock interconnect accounts for the vast majority of the delays associated with signals as they propagate through an FPGA.

Insider Info

In addition to implementing glue logic and irregular structures like state machines, fine-grained architectures are said to be particularly efficient when executing systolic algorithms (functions that benefit from massively parallel implementations). These architectures are also said to offer some advantages with regard to traditional logic synthesis technology, which is geared toward fine-grained ASIC architectures.

LOGIC BLOCKS

There are two fundamental incarnations of the programmable logic blocks used to form the medium-grained architectures referenced in the previous section: MUX (multiplexer) based and LUT (lookup table) based.

MUX-based

As an example of a MUX-based approach, consider one way in which the 3-input function $y = (a \,\&\, b) \mid c$ could be implemented using a block containing only multiplexers (Figure 2-2).

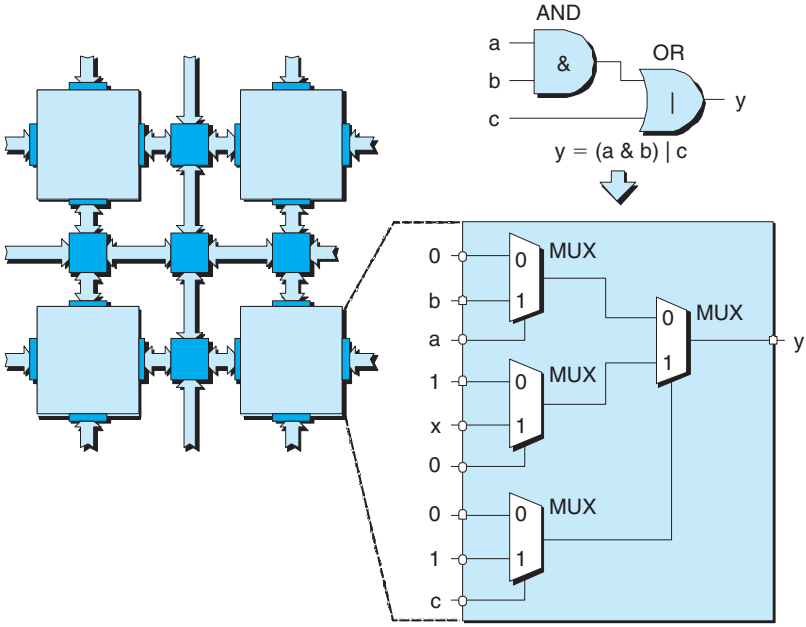


FIGURE 2-2 MUX-based logic block.

The device can be programmed such that each input to the block is presented with a logic 0, a logic 1, or the true or inverse version of a signal (*a*, *b*, or *c* in this case) coming from another block or from a primary input to the device. This allows each block to be configured in myriad ways to implement a plethora of possible functions. (The *x* shown on the input to the central multiplexer in Figure 2-2 indicates that we don't care whether this input is connected to a 0 or a 1.)

LUT-based

The underlying concept behind a LUT is relatively simple. A group of input signals is used as an index (pointer) to a lookup table. The contents of this table are arranged such that the cell pointed to by each input combination contains the desired value. For example, let's assume that we wish to implement the function:

$$y = (a \ \& \ b) | c$$

This can be achieved by loading a 3-input LUT with the appropriate values. For the purposes of the following examples, we shall assume that the LUT is formed from SRAM cells (but it could be formed using antifuses, E2PROM, or FLASH cells, as discussed earlier in this chapter). A commonly used technique is to use the inputs to select the desired SRAM cell using a cascade of transmission gates as shown in Figure 2-3. (Note that the SRAM cells will also

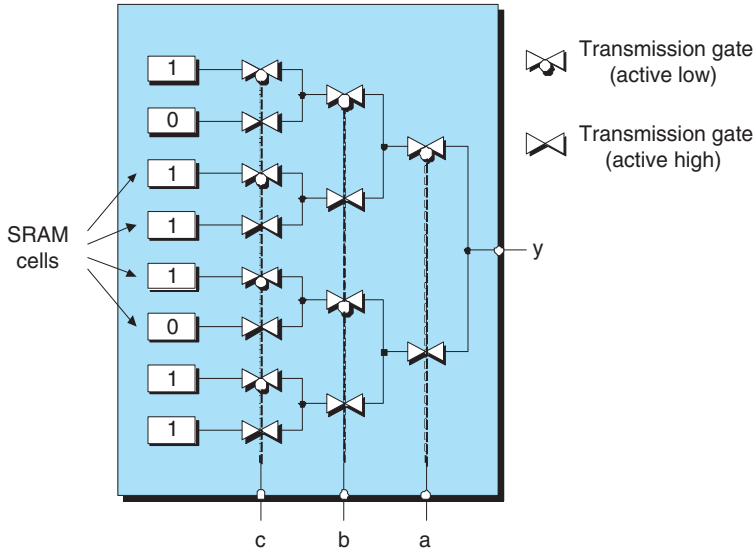


FIGURE 2-3 A transmission gate-based LUT (programming chain omitted for purposes of clarity).

be connected together in a chain for configuration purposes—that is, to load them with the required values—but these connections have been omitted from this illustration to keep things simple.)

If a transmission gate is enabled (active), it passes the signal seen on its input through to its output. If the gate is disabled, its output is electrically disconnected from the wire it is driving.

The transmission gate symbols shown with a small circle (called a "bobble" or a "bubble") indicate that these gates will be activated by a logic 0 on their control input. By comparison, symbols without bobbles indicate that these gates will be activated by a logic 1. Based on this understanding, it's easy to see how different input combinations can be used to select the contents of the various SRAM cells.

—Technology Trade-offs—

- If you take a group of logic gates several layers deep, then a LUT approach can be very efficient in terms of resource utilization and input-to-output delays. (In this context, "deep" refers to the number of logic gates between the inputs and the outputs. Thus, the function illustrated in Figure 2-4 would be said to be two layers deep.) However, one downside to a LUT-based architecture is that if you only want to implement a small function—such as a 2-input AND gate—somewhere in your design, you'll end up using an entire LUT to do so. In addition to being wasteful in terms of resources, the resulting delays are high for such a simple function.
- By comparison, in the case of MUX-based architectures containing a mixture of MUXes and logic gates, it's often possible to gain access to

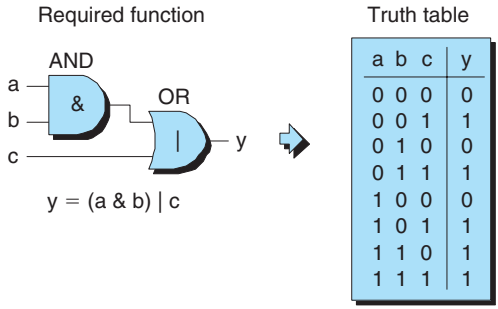


FIGURE 2-4 Required function and associated truth table.

intermediate values from the signals linking the logic gates and the MUXes. In this case, each logic block can be broken down into smaller fragments, each of which can be used to implement a simple function. Thus, these architectures may offer advantages in terms of performance and silicon utilization for designs containing large numbers of independent simple logic functions.

- It is said that MUX-based architectures have an advantage when it comes to implementing control logic along the lines of "if this input is *true* and this input is *false*, then make that output *true*…" However, some of these architectures don't provide high speed carry logic chains, in which case their LUT-based counterparts are left as the leaders in anything to do with arithmetic processing.

Insider Info

In the past, some devices were created using a mixture of different LUT sizes, such as 3-input and 4-input LUTs, because this offered the promise of optimal device utilization. However, one of the main tools in the design engineer's treasure chest is logic synthesis, and uniformity and regularity are what a synthesis tool likes best. Thus, all the really successful architectures are currently based only on the use of 4-input LUTs. (This is not to say that mixed-size LUT architectures won't reemerge in the future as design software continues to increase in sophistication.)

LUT versus Distributed RAM versus SR

The fact that the core of a LUT in an SRAM-based device comprises a number of SRAM cells offers some interesting possibilities. In addition to its primary role as a lookup table, some vendors allow the cells forming the LUT to be used as a small block of RAM (the 16 cells forming a 4-input LUT, for example, could be cast in the role of a 16 × 1 RAM). This is referred to as

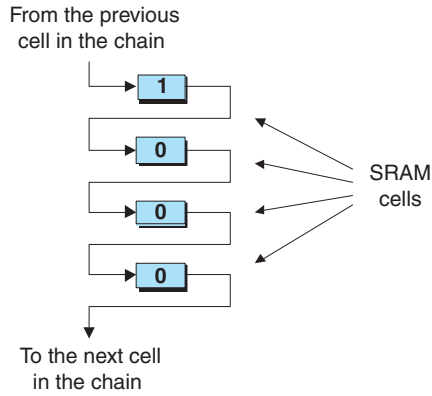


FIGURE 2-5 Configuration cells linked in a chain.

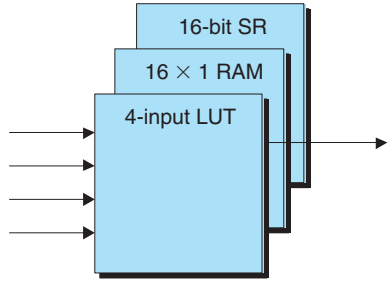


FIGURE 2-6 A multifaceted LUT.

distributed RAM because (a) the LUTs are strewn (distributed) across the surface of the chip, and (b) this differentiates it from the larger chunks of block RAM (introduced later in this chapter).

Yet another possibility devolves from the fact that all of the FPGA's configuration cells—including those forming the LUT—are effectively strung together in a long chain (Figure 2-5).

This aspect of the architecture is discussed in more detail in Chapter 3. The point here is that, once the device has been programmed, some vendors allow the SRAM cells forming a LUT to be treated independently of the main body of the chain and to be used in the form of a shift register. Thus, each LUT may be considered multifaceted (Figure 2-6).

CLBs VERSUS LABs VERSUS SLICES

In addition to one or more LUTs, a programmable logic block will contain other elements, such as multiplexers and registers. But before we delve into this topic, we first need to wrap our brains around some terminology.

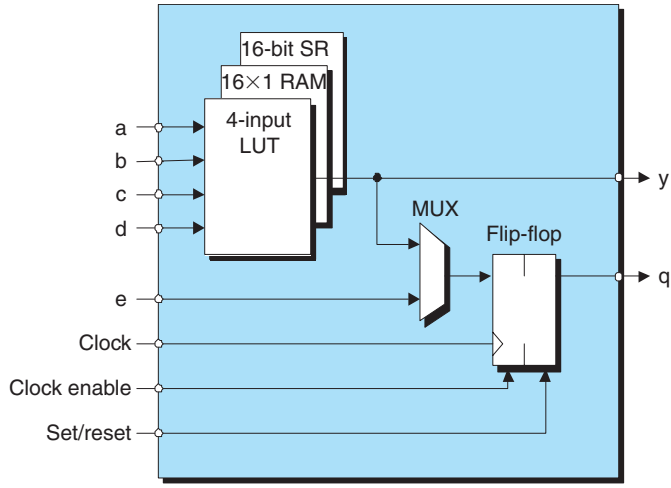


FIGURE 2-7 A simplified view of a Xilinx LC.

Logic Cells/Logic Elements

Each FPGA vendor has its own names for things. For example, the core building block in a modern FPGA from Xilinx is called a *logic cell* (LC). Among other things, an LC comprises a 4-input LUT (which can also act as a 16 × 1 RAM or a 16-bit shift register), a multiplexer, and a register (Figure 2-7).

How It Works

The illustration presented in Figure 2-7 is a gross simplification, but it serves our purposes here. The register can be configured to act as a flip-flop, as shown in Figure 2-7, or as a latch. The polarity of the clock (rising-edge triggered or falling-edge triggered) can be configured, as can the polarity of the clock enable and set/reset signals (active-high or active-low). In addition to the LUT, MUX, and register, the LC also contains a smattering of other elements, including some special fast carry logic for use in arithmetic operations (this is discussed in more detail a little later).

Just for reference, the equivalent core building block in an FPGA from Altera is called a ***logic element*** *(LE). There are a number of differences between a Xilinx LC and an Altera LE, but the overall concepts are very similar.*

Slicing and Dicing

The next step up the hierarchy is what Xilinx calls a slice (Altera and the other vendors have their own equivalent names). At the time of this writing, a slice contains two logic cells (Figure 2-8).

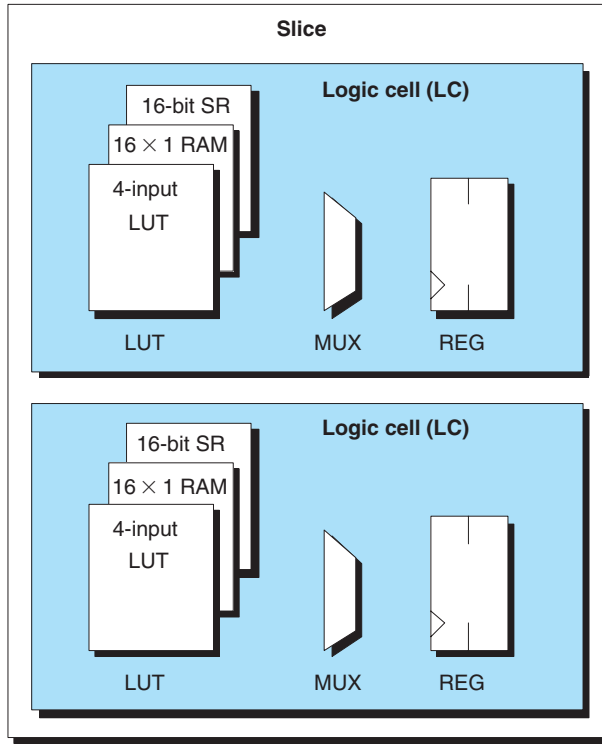


FIGURE 2-8 A slice containing two logic cells.

The internal wires have been omitted from this illustration to keep things simple; it should be noted, however, that although each logic cell's LUT, MUX, and register have their own data inputs and outputs, the slice has one set of clock, clock enable, and set/reset signals common to both logic cells.

CLBs and LABs

And moving one more level up the hierarchy, we come to what Xilinx calls a *configurable logic block* (CLB) and what Altera refers to as a *logic array block* (LAB).

Using CLBs as an example, some Xilinx FPGAs have two slices in each CLB, while others have four. At the time of this writing, a CLB equates to a single logic block in our original visualization of "islands" of programmable logic in a "sea" of programmable interconnect (Figure 2-9).

There is also some fast programmable interconnect within the CLB. This interconnect (not shown in Figure 2-9 for reasons of clarity) is used to connect neighboring slices.

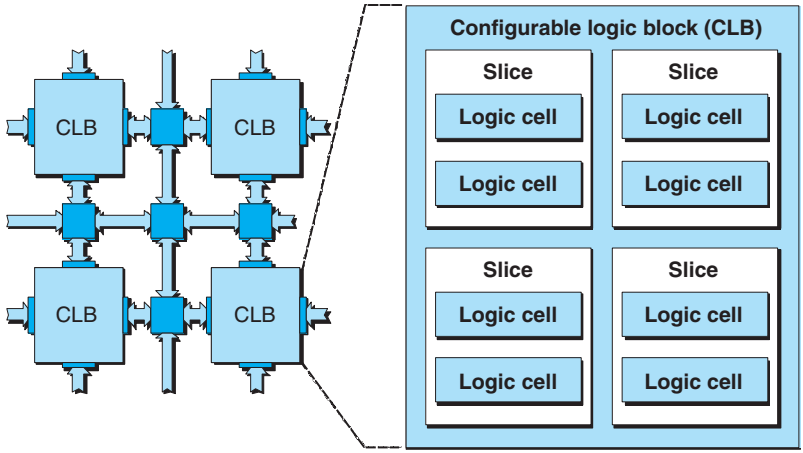


FIGURE 2-9 A CLB containing four slices (the number of slices depends on the FPGA family).

Insider Info

The reason for having this type of logic-block hierarchy—LC→Slice (with two LCs)→CLB (with four slices)—is that it is complemented by an equivalent hierarchy in the interconnect. Thus, there is fast interconnect between the LCs in a slice, then slightly slower interconnect between slices in a CLB, followed by the interconnect between CLBs. The idea is to achieve the optimum trade-off between making it easy to connect things together without incurring excessive interconnect-related delays.

Distributed RAMs and Shift Registers

We previously noted that each 4-bit LUT can be used as a 16 × 1 RAM. And things just keep on getting better because, assuming the four-slices-per-CLB configuration illustrated in Figure 2-9, all of the LUTs within a CLB can be configured together to implement the following:

- Single-port 16 × 8 bit RAM
- Single-port 32 × 4 bit RAM
- Single-port 64 × 2 bit RAM
- Single-port 128 × 1 bit RAM
- Dual-port 16 × 4 bit RAM
- Dual-port 32 × 2 bit RAM
- Dual-port 64 × 1 bit RAM

Alternatively, each 4-bit LUT can be used as a 16-bit shift register. In this case, there are special dedicated connections between the logic cells within a slice and between the slices themselves that allow the last bit of one shift register to

be connected to the first bit of another without using the ordinary LUT output (which can be used to view the contents of a selected bit within that 16-bit register). This allows the LUTs within a single CLB to be configured together to implement a shift register containing up to 128 bits as required.

FAQs

***What is a* fast carry chain?**

A key feature of modern FPGAs is that they include the special logic and interconnect required to implement fast carry chains. In the context of the CLBs introduced in the previous section, each LC contains special carry logic. This is complemented by dedicated interconnect between the two LCs in each slice, between the slices in each CLB, and between the CLBs themselves. This special carry logic and dedicated routing boosts the performance of logical functions such as counters and arithmetic functions such as adders. The availability of these fast carry chains—in conjunction with features like the shift register incarnations of LUTs (discussed previously) and embedded multipliers and the like (introduced in following sections)—provided the wherewithal for FPGAs to be used for applications like DSP.

EMBEDDED RAMs

Many applications require the use of memory, so FPGAs now include relatively large chunks of embedded RAM called *e-RAM* or *block RAM*. Depending on the architecture of the component, these blocks might be positioned around the periphery of the device, scattered across the face of the chip in relative isolation, or organized in columns, as shown in Figure 2-10.

Depending on the device, such a RAM might be able to hold anywhere from a few thousand to tens of thousands of bits. Furthermore, a device might contain anywhere from tens to hundreds of these RAM blocks, thereby providing a total storage capacity of a few hundred thousand bits all the way up to several million bits.

Each block of RAM can be used independently, or multiple blocks can be combined together to implement larger blocks. These blocks can be used for a variety of purposes, such as implementing standard single- or dual-port RAMs, first-in first-out (FIFO) functions, state machines, and so forth.

EMBEDDED MULTIPLIERS, ADDERS, ETC.

Some functions, like multipliers, are inherently slow if they are implemented by connecting a large number of programmable logic blocks together. Since many applications require these functions, many FPGAs incorporate special hardwired multiplier blocks. These are typically located in

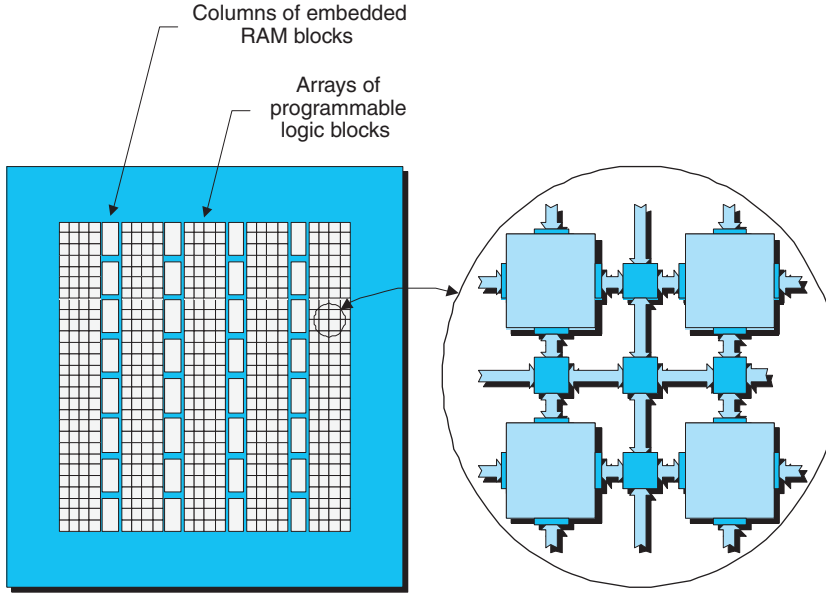


FIGURE 2-10 Bird's-eye view of chip with columns of embedded RAM blocks.

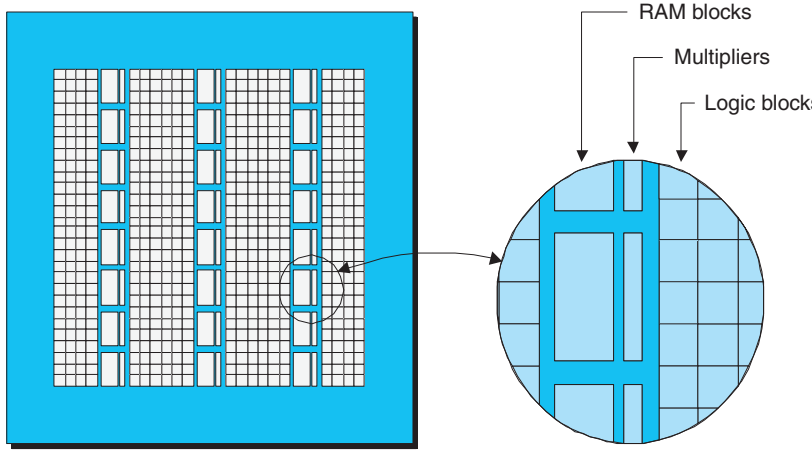


FIGURE 2-11 Bird's-eye view of chip with columns of embedded multipliers and RAM blocks.

close proximity to the embedded RAM blocks introduced in the previous point because these functions are often used in conjunction with each other (Figure 2-11).

Similarly, some FPGAs offer dedicated adder blocks. One operation very common in DSP-type applications is called a multiply-and-accumulate

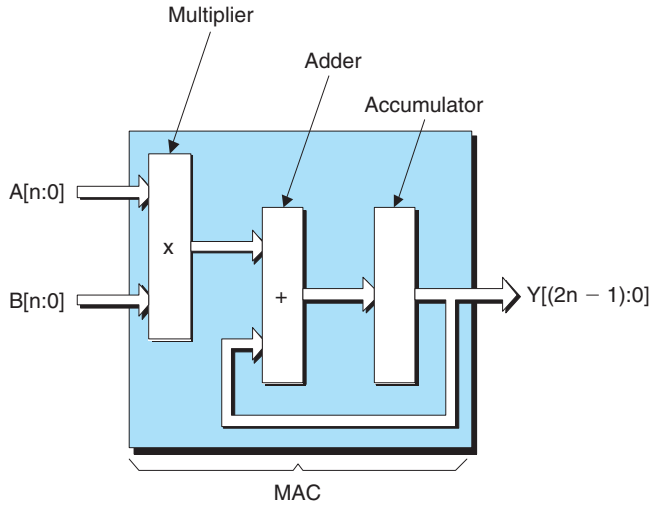


FIGURE 2-12 The functions forming a MAC.

(MAC) (Figure 2-12). As its name would suggest, this function multiplies two numbers together and adds the result to a running total stored in an accumulator.

Key Concept

If the FPGA you are working with supplies only embedded multipliers, you will have to implement this function by combining the multiplier with an adder formed from a number of programmable logic blocks, while the result is stored in some associated flip-flops, in a block RAM, or in a number of distributed RAMs. Life becomes a little easier if the FPGA also provides embedded adders, and some FPGAs provide entire MACs as embedded functions.

EMBEDDED PROCESSOR CORES

Almost any portion of an electronic design can be realized in hardware (using logic gates and registers, etc.) or software (as instructions to be executed on a microprocessor). One of the main partitioning criteria is how fast you wish the various functions to perform their tasks:

- Picosecond and nanosecond logic: This has to run insanely fast, which mandates that it be implemented in hardware (in the FPGA fabric).
- Microsecond logic: This is reasonably fast and can be implemented either in hardware or software (this type of logic is where you spend the bulk of your time deciding which way to go).
- Millisecond logic: This is the logic used to implement interfaces such as reading switch positions and flashing light-emitting diodes (LEDs). It's a

pain slowing the hardware down to implement this sort of function (using huge counters to generate delays, for example). Thus, it's often better to implement these tasks as microprocessor code (because processors give you lousy speed—compared to dedicated hardware—but fantastic complexity).

The fact is that the majority of designs make use of microprocessors in one form or another. Until recently, these appeared as discrete devices on the circuit board. Of late, high-end FPGAs have become available that contain one or more embedded microprocessors, which are typically referred to as microprocessor cores. In this case, it often makes sense to move all of the tasks that used to be performed by the external microprocessor into the internal core. This provides a number of advantages, not the least being that it saves the cost of having two devices; it eliminates large numbers of tracks, pads, and pins on the circuit board; and it makes the board smaller and lighter.

Hard Microprocessor Cores

A hard microprocessor core is implemented as a dedicated, predefined block. There are two main approaches for integrating such a core into the FPGA:

1. Locate it in a strip (actually called "The Stripe") to the side of the main FPGA fabric (Figure 2-13). In this scenario, all of the components are typically formed on the same silicon chip, although they could also be formed on two chips and packaged as a multichip module (MCM). The main FPGA fabric would also include the embedded RAM blocks, multipliers, and so on, but these have been omitted from this illustration to keep things simpler.

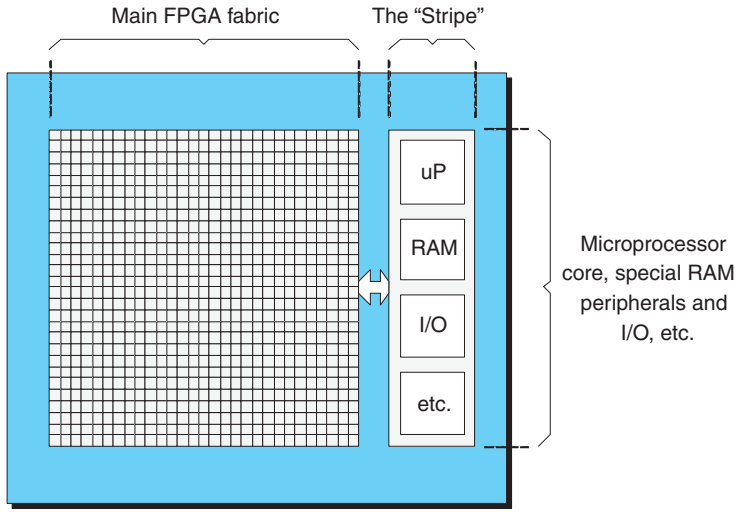


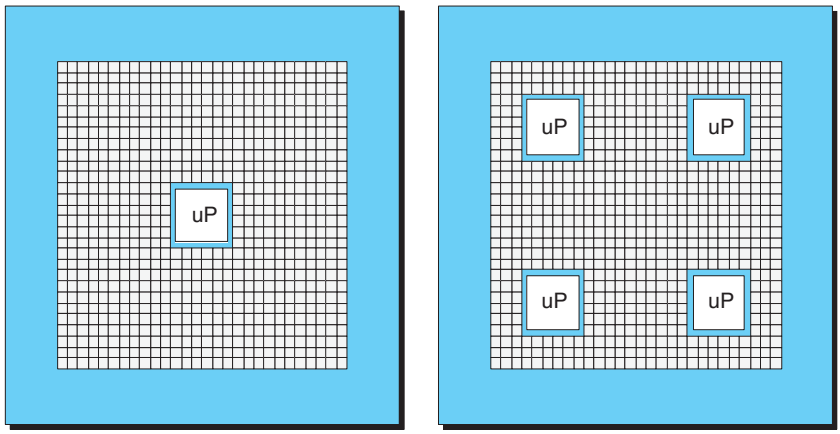
FIGURE 2-13 Bird's-eye view of chip with embedded core outside of the main fabric.

One advantage of this implementation is that the main FPGA fabric is identical for devices with and without the embedded microprocessor core, which can help make things easier for the design tools used by the engineers. The other advantage is that the FPGA vendor can bundle a whole load of additional functions in the strip to complement the microprocessor core, such as memory, special peripherals, and so forth.

2. An alternative is to embed one or more microprocessor cores directly into the main FPGA fabric. One-, two-, and even four-core implementations are currently available (Figure 2-14). In this case, the design tools have to be able to take account of the presence of these blocks in the fabric; any memory used by the core is formed from embedded RAM blocks, and any peripheral functions are formed from groups of general-purpose programmable logic blocks. Proponents of this scheme will argue that there are inherent speed advantages to be gained from having the microprocessor core in intimate proximity to the main FPGA fabric.

Soft Microprocessor Cores

As opposed to embedding a microprocessor physically into the fabric of the chip, it is possible to configure a group of programmable logic blocks to act as a microprocessor. These are typically called soft cores, but they may be more precisely categorized as either "soft" or "firm" depending on the way in which the microprocessor's functionality is mapped onto the logic blocks. Soft cores are simpler (more primitive) and slower than their hard-core counterparts.



(a) One embedded core (b) Four embedded cores

FIGURE 2-14 Bird's-eye view of chips with embedded cores inside the main fabric.

—Technology Trade-offs—

- *A soft core typically runs at 30 to 50 percent of the speed of a hard core.*
- *However, they have the advantage that you only need to implement a core if you need it and that you can instantiate as many cores as you require until you run out of resources in the form of programmable logic blocks.*

CLOCK MANAGERS

All of the synchronous elements inside an FPGA—for example, the registers configured to act as flip-flops inside the programmable logic blocks—need to be driven by a clock signal. Such a clock signal typically originates in the outside world, comes into the FPGA via a special clock input pin, and is then routed through the device and connected to the appropriate registers.

Clock Trees

Consider a simplified representation that omits the programmable logic blocks and shows only the clock tree and the registers to which it is connected (Figure 2-15).

This is called a *clock tree* because the main clock signal branches again and again (the flip-flops can be considered the "leaves" on the end of the branches). This structure is used to ensure that all of the flip-flops see their versions of the clock signal as close together as possible. If the clock were distributed as a single long track driving all of the flip-flops one after another, then the flip-flop closest to the clock pin would see the clock signal much sooner than the one at the end of the chain. This is referred to as skew, and it

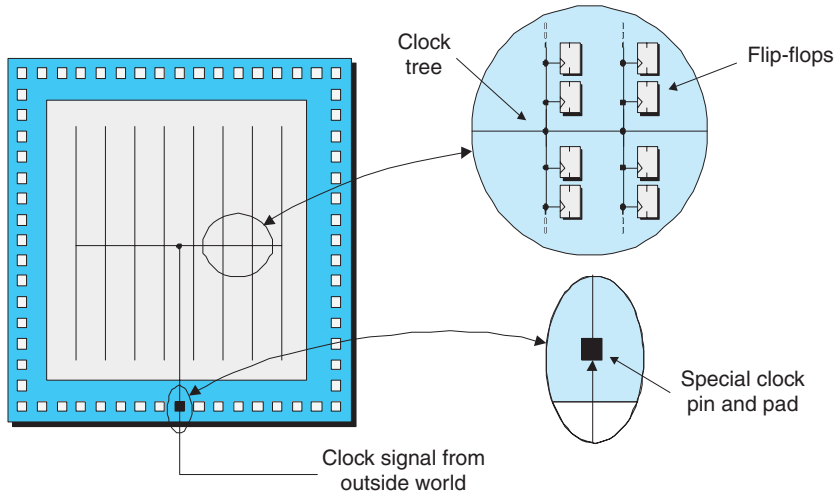


FIGURE 2-15 A simple clock tree.

can cause all sorts of problems (even when using a clock tree, there will be a certain amount of skew between the registers on a branch and between branches). The clock tree is implemented using special tracks and is separate from the general-purpose programmable interconnect. The scenario shown above is actually very simplistic.

Clock Managers

Instead of configuring a clock pin to connect directly into an internal clock tree, that pin can be used to drive a special hard-wired function (block) called a *clock manager* that generates a number of daughter clocks (Figure 2-16).

These daughter clocks may be used to drive internal clock trees or external output pins that can be used to provide clocking services to other devices on the host circuit board. Each family of FPGAs has its own type of clock manager (there may be multiple clock manager blocks in a device), where different clock managers may support only a subset of the following features:

Jitter removal: For the purposes of a simple example, assume that the clock signal has a frequency of 1 MHz (in reality, of course, this could be much, much higher). In an ideal environment each clock edge from the outside world would arrive exactly 1 millionth of a second after its predecessor. In the real world, however, clock edges may arrive a little early or a little late. As one way to visualize this effect—known as *jitter*—imagine if we were to superimpose multiple edges on top of each other; the result would be a "fuzzy" clock (Figure 2-17). The FPGA's clock manager can be used to detect and correct for this jitter and to provide "clean" daughter clock signals for use inside the device (Figure 2-18).

Frequency synthesis: It may be that the frequency of the clock signal being presented to the FPGA from the outside world is not exactly what the design engineers wish for. In this case, the clock manager can be used to generate daughter clocks with frequencies that are derived by multiplying or dividing the original signal. As a really simple example, consider three daughter clock signals: the first with a frequency equal to that of the original clock, the second multiplied to be twice that of the original clock, and the third divided to be

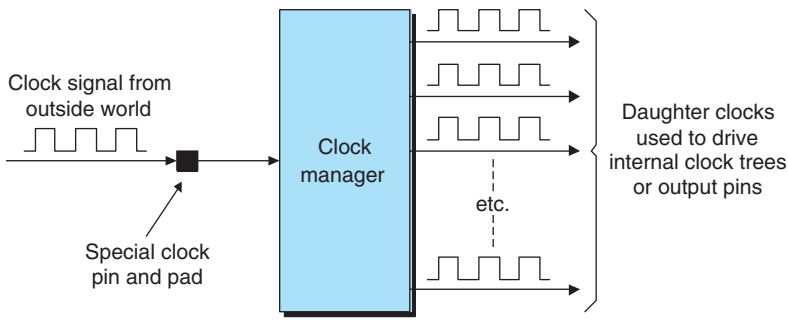


FIGURE 2-16 A clock manager generates daughter clocks.

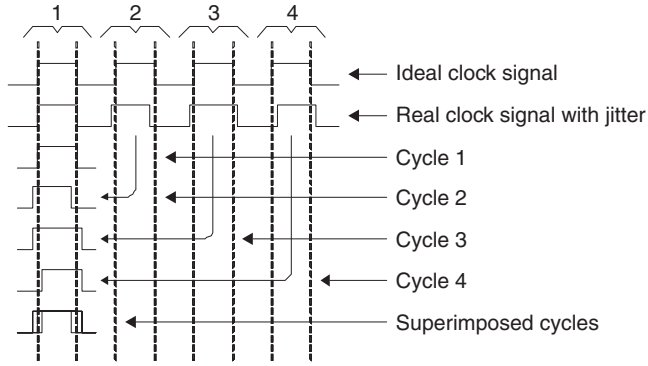


FIGURE 2-17 Jitter results in a fuzzy clock.

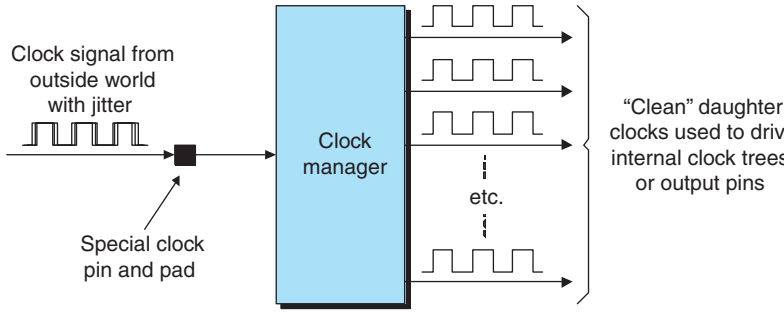

Clock signal from outside world with jitter
Special clock pin and pad
Clock manager
etc.
"Clean" daughter clocks used to drive internal clock trees or output pins

FIGURE 2-18 The clock manager can remove jitter.

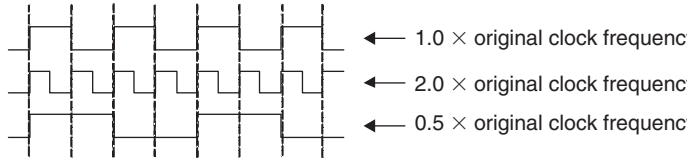


FIGURE 2-19 Using the clock manager to perform frequency synthesis.

half that of the original clock (Figure 2-19). Once again, Figure 2-19 reflects very simple examples. In the real world, one can synthesize all sorts of internal clocks, such as an output that is four-fifths the frequency of the original clock.

Phase shifting: Certain designs require the use of clocks that are phase shifted (delayed) with respect to each other. Some clock managers allow you to select from fixed phase shifts of common values such as 120° and 240° (for a three-phase clocking scheme) or 90°, 180°, and 270° (if a four-phase clocking scheme is required). Others allow you to configure the exact amount of phase shift you require for each daughter clock. For example, let's assume that we are deriving four internal clocks from a master clock, where the first is in

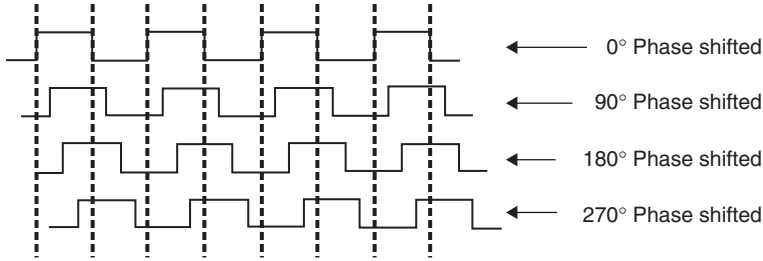


FIGURE 2-20 Using the clock manager to phase-shift the daughter clocks.

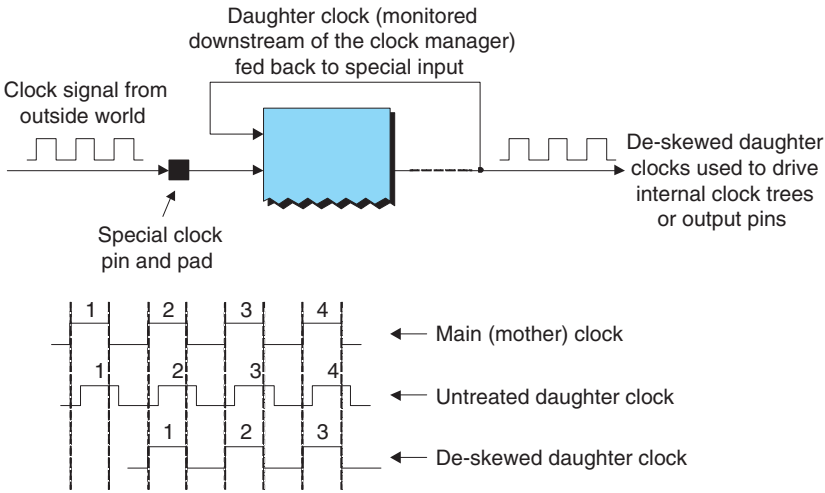

Daughter clock (monitored downstream of the clock manager) fed back to special input
Clock signal from outside world
De-skewed daughter clocks used to drive internal clock trees or output pins
Special clock pin and pad
1 2 3 4
Main (mother) clock
1 2 3 4
Untreated daughter clock
1 2 3
De-skewed daughter clock

FIGURE 2-21 Deskewing with reference to the mother clock.

phase with the original clock, the second is phase shifted by 90°, the third by 180°, and so forth (Figure 2-20).

Auto-skew correction: For the sake of simplicity, let's assume that we're talking about a daughter clock that has been configured to have the same frequency and phase as the main clock signal coming into the FPGA. By default, however, the clock manager will add some element of delay to the signal as it performs its machinations. Also, more significant delays will be added by the driving gates and interconnect employed in the clock's distribution. The result is that—if nothing is done to correct it—the daughter clock will lag behind the input clock by some amount. Once again, the difference between the two signals is known as *skew*. Depending on how the main clock and the daughter clock are used in the FPGA (and on the rest of the circuit board), this can cause a variety of problems. Thus, the clock manager may allow a special input to feed the daughter clock. In this case, the clock manager will compare the two signals and specifically add additional delay to the daughter clock sufficient to realign it with the main clock (Figure 2-21).

To be a tad more specific, only the *prime* (zero phase-shifted) daughter clock will be treated in this way, and all of the other daughter clocks will be phase aligned to this prime daughter clock.

—Technology Trade-offs—

- *Some FPGA clock managers are based on phase-locked loops (PLLs), while others are based on digital delay-locked loops (DLLs). PLLs have been used since the 1940s in analog implementations, but recent emphasis on digital methods has made it desirable to match signal phases digitally. PLLs can be implemented using either analog or digital techniques, while DLLs are by definition digital in nature.*
- *The proponents of DLLs say that they offer advantages in terms of precision, stability, power management, noise insensitivity, and jitter performance.*

GENERAL-PURPOSE I/O

Today's FPGA packages can have a thousand or more pins, which are arranged as an array across the base of the package. Similarly, when it comes to the silicon chip inside the package, flip-chip packaging strategies allow the power, ground, clock, and I/O pins to be presented across the surface of the chip. Purely for the purposes of these discussions (and illustrations), however, it makes things simpler if we assume that all of the connections to the chip are presented in a ring around the circumference of the device, as indeed they were for many years.

Configurable I/O Standards

Let's consider for a moment an electronic product from the perspective of the architects and engineers designing the circuit board. Depending on what they are trying to do, the devices they are using, the environment the board will operate in, and so on, these designers will select a particular standard to be used to transfer data signals. (In this context, "standard" refers to electrical aspects of the signals, such as their logic 0 and logic 1 voltage levels.) The problem is that there is a wide variety of such standards, and it would be painful to have to create special FPGAs to accommodate each variation. For this reason, an FPGA's general-purpose I/O can be configured to accept and generate signals conforming to whichever standard is required. These general-purpose I/O signals will be split into a number of banks—we'll assume eight such banks numbered from 0 to 7 (Figure 2-22).

The interesting point is that each bank can be configured individually to support a particular I/O standard. Thus, in addition to allowing the FPGA to work with devices using multiple I/O standards, this allows the FPGA to actually be used to interface between different I/O standards (and to translate between different protocols that may be based on particular electrical standards).

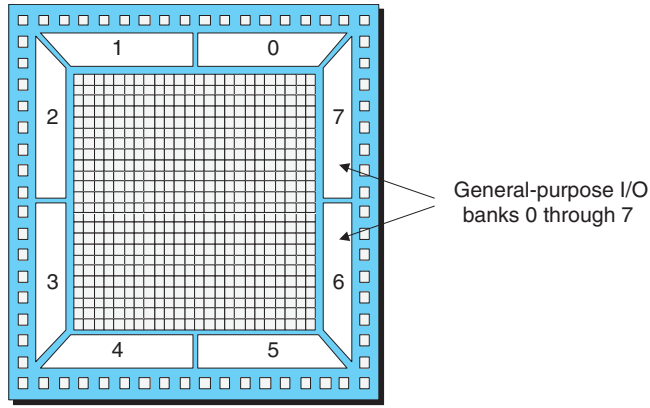


FIGURE 2-22 Bird's-eye view of chip showing general-purpose I/O banks.

Configurable I/O Impedances

The signals used to connect devices on today's circuit board often have fast edge rates (this refers to the time it takes the signal to switch between one logic value and another). In order to prevent signals reflecting back (bouncing around), it is necessary to apply appropriate terminating resistors to the FPGA's input or output pins. In the past, these resistors were applied as discrete components that were attached to the circuit board outside the FPGA. However, this technique became increasingly problematic as the number of pins started to increase and their pitch (the distance between them) shrank. For this reason, today's FPGAs allow the use of internal terminating resistors whose values can be configured by the user to accommodate different circuit board environments and I/O standards.

Core versus I/O Supply Voltages

TABLE 2-1 Supply Voltages versus Technology Nodes

Year	Supply (Core Voltage (V))	Technology Node (nm)
1998	3.3	350
1999	2.5	250
2000	1.8	180
2001	1.5	150
2003	1.2	130

The supply voltage (which is actually provided using large numbers of power and ground pins) is used to power the FPGA's internal logic. For this reason, this is known as the *core voltage*. However, different I/O standards may use signals with voltage levels significantly different from the core voltage, so each bank of general-purpose I/Os can have its own additional supply pins.

Insider Info

It's interesting to note that—from the 350 nm node onward—the core voltage has scaled fairly linearly with the process technology. However, there are physical reasons not to go much below 1 V (these reasons are based on technology aspects such as transistor input switching thresholds and voltage drops), so this "voltage staircase" might start to tail off in the not-so-distant future.

GIGABIT TRANSCEIVERS

The traditional way to move large amounts of data between devices is to use a bus, a collection of signals that carry similar data and perform a common function (Figure 2-23). Early microprocessor-based systems circa 1975 used 8-bit buses to pass data around. As the need to push more data around and to move it faster grew, buses grew to 16 bits in width, then 32 bits, then 64 bits, and so forth. The problem is that this requires a lot of pins on the device and a lot of tracks connecting the devices together. Routing these tracks so that they all have the same length and impedance becomes increasingly painful as boards grow in complexity. Furthermore, it becomes increasingly difficult to manage signal integrity issues (such as susceptibility to noise) when you are dealing with large numbers of bus-based tracks.

For this reason, today's high-end FPGAs include special hard-wired gigabit transceiver blocks. These blocks use one pair of *differential signals* (which means a pair of signals that always carry opposite logical values) to transmit (TX) data and another pair to receive (RX) data (Figure 2-24).

These transceivers operate at incredibly high speeds, allowing them to transmit and receive billions of bits of data per second. Furthermore, each block actually supports a number (say four) of such transceivers, and an FPGA

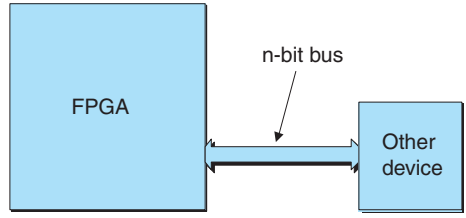


FIGURE 2-23 Using a bus to communicate between devices.

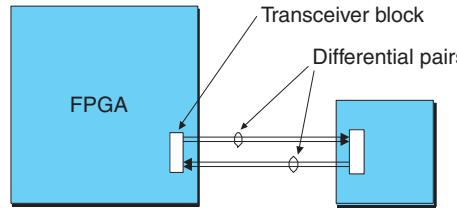


FIGURE 2-24 Using high-speed transceivers to communicate between devices.

may contain a number of these transceiver blocks. At the time of this writing only a few percent of designs make use of these transceivers, but this number is expected to rise dramatically over the next few years. Using these gigabit transceivers is something of an art form, but each FPGA vendor will provide detailed user guides and application notes for its particular technology.

Multiple Standards

Of course, electronics wouldn't be electronics if there weren't a variety of standards for this sort of thing. Each standard defines things from the high-level protocols on down to the *physical layer* (PHY). A few of the more common standards are:

- Fibre Channel
- InfiniBand®
- PCI Express
- RapidIO™
- SkyRail™ (from MindSpeed Technologies)
- 10-gigabit Ethernet

This situation is further complicated by the fact that, in the case of some of these standards, like PCI Express and SkyRail, device vendors might use the same underlying concepts, but rebrand things using their own names and terminology. Also, implementing some standards requires the use of multiple transceiver blocks.

—Technology Trade-offs—

- *Let's assume that we're building a circuit board and wish to use some form of high-speed serial interface. In this case, the system architects will determine which standard is to be used. Each of the gigabit transceiver blocks in an FPGA can generally be configured to support a number of different standards, but usually not all of them. This means that the system architects will either select a standard that is supported by the FPGAs they intend to use, or they will select FPGAs that will support the interface standard they wish to employ.*

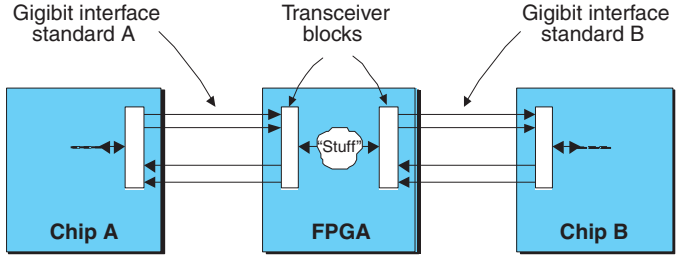


FIGURE 2-25 Using an FPGA to interface between multiple standards.

- *If the system under consideration includes creating one or more ASICs, we can of course implement the standard of our choice from the ground up (or more likely we would purchase an appropriate block of IP from a third-party vendor). Off-the-shelf (ASSP-type) devices, however, will typically support only one, or a subset, of the above standards. In this case, an FPGA may be used to act as an interface between two (or more) standards (Figure 2-25).*

INTELLECTUAL PROPERTY (IP)

Today's FPGA designs are so big and complex that it would be impractical to create every portion of the design from scratch. One solution is to reuse existing functional blocks for the boring stuff and spend the bulk of your time and resources creating the new portions of the design that will differentiate your design from any competing offerings.

Any existing functional blocks are typically referred to as *intellectual property* (IP). The three main sources of such IP are:

1. internally created blocks reused from previous designs,
2. FPGA vendors, and
3. third-party IP providers.

For the purposes of these discussions, we shall concentrate on the latter two categories.

Each FPGA vendor offers its own selection of hard, firm, and soft IP. *Hard IP* comes in the form of preimplemented blocks such as microprocessor cores, gigabit interfaces, multipliers, adders, MAC functions, and the like. These blocks are designed to be as efficient as possible in terms of power consumption, silicon real estate, and performance. Each FPGA family will feature different combinations of such blocks, together with various quantities of programmable logic blocks.

At the other end of the spectrum, *soft IP* refers to a source-level library of high-level functions that can be included to the users' designs. These functions are typically represented using a hardware description language, or HDL, such

as Verilog or VHDL at the register transfer level (RTL) of abstraction. Any soft IP functions the design engineers decide to use are incorporated into the main body of the design—which is also specified in RTL—and subsequently synthesized down into a group of programmable logic blocks (possibly combined with some hard IP blocks like multipliers, etc.).

Holding somewhat of a middle ground is *firm IP*, which also comes in the form of a library of high-level functions. Unlike their soft IP equivalents, however, these functions have already been optimally mapped, placed, and routed into a group of programmable logic blocks (possibly combined with some hard IP blocks like multipliers, etc.). One or more copies of each predefined firm IP block can be instantiated (called up) into the design as required.

FAQs

How do you decide whether to use hard, firm, or soft IP?

It can be hard to draw the line between those functions that are best implemented as hard IP and those that should be implemented as soft or firm IP. In the case of functions like the multipliers, adders, and MACs discussed earlier in this chapter, these are generally useful for a wide range of applications. On the other hand, some FPGAs contain dedicated blocks to handle specific interface protocols like the PCI standard. It can, of course, make your life much easier if this happens to be the interface you wish to use to connect your device to the rest of the board. On the other hand, if you decide you need to use some other interface, a dedicated PCI block will serve only to waste space, block traffic, and burn power in your chip.

Some IP that used to be "soft" is now becoming "hard." For example, the most current generation of FPGAs contains hard processor, clock manager, Ethernet, and gigabit I/O blocks, among others. These help bring high-end ASIC functionality into standard FPGAs. Over time, it is likely that additional functions of this ilk will be incorporated into the FPGA device.

Insider Info

Generally speaking, once FPGA vendors add a function like this into their device, they've essentially placed the component into a niche. Sometimes this must be done to achieve the desired performance, but this is a classic problem because the next generation of the device is often fast enough to perform this function in its main (programmable) fabric.

Handcrafted IP

One scenario is that the IP provider has handcrafted an IP block starting with an RTL description (the provider might also have used an IP block/core

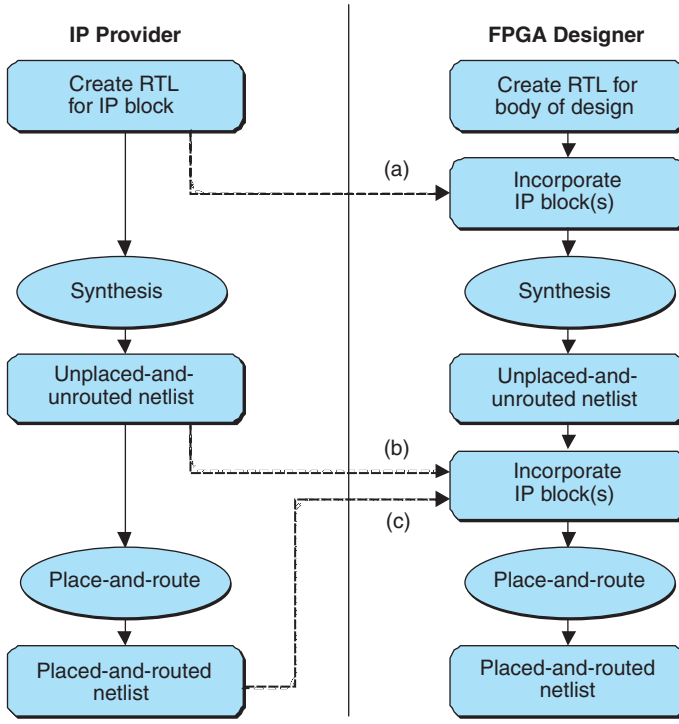


FIGURE 2-26 Alternative potential IP acquisition points.

generator application, as discussed later in this chapter). In this case, there are several ways in which the end user might purchase and use such a block (Figure 2-26):

1. *Blocks of unencrypted source code*. These blocks can then be integrated into the RTL code for the body of the design (Figure 2-26a). (Note that the IP provider would already have simulated, synthesized, and verified the IP blocks before handing over the RTL source code.)

Insider Info

Generally speaking, this is an expensive option because IP providers typically don't want anyone to see their RTL source code. Certainly, FPGA vendors are usually reluctant to provide unencrypted RTL because they don't want anyone to retarget it toward a competitor's device offering. So if you really wish to go this route, whoever is providing the IP will charge you an arm and a leg, and you'll end up signing all sorts of licensing and nondisclosure agreements (NDAs).

2. *Encrypted RTL level*. Unfortunately, at the time of this writing, there is no industry-standard encryption technique for RTL that has popular tool support. This has led companies like Altera and Xilinx to develop their own encryption schemes and tools. RTL encrypted by a particular FPGA vendor's tools can only be processed by that vendor's own synthesis tools (or sometimes by a third-party synthesis tool that has been OEM'd by the FPGA vendor).
3. *Unplaced-and-unrouted netlist level*. Perhaps the most common scenario is for FPGA designers to purchase IP at the unplaced-and-unrouted LUT/CLB netlist level (Figure 2-26b). Such netlists are typically provided in encrypted form, either as encrypted EDIF or using some FPGA vendor-specific format. In this case, the IP vendor may also provide a compiled cycle-accurate C/C++ model to be used for functional verification because such a model will simulate much faster than the LUT/CLB netlist-level model.

—Technology Trade-offs—

- *The main advantage of this scenario is that the IP provider has often gone to a lot of effort tuning the synthesis engine and handcrafting certain portions of the function to achieve an optimal implementation in terms of resource utilization and performance.*
- *One disadvantage is that the FPGA designer doesn't have any ability to remove unwanted functionality.*
- *Another disadvantage is that the IP block is tied to a particular FPGA vendor and device family.*

4. *Placed-and-routed netlist level.* In certain cases, the FPGA designer may purchase IP at the placed-and-routed LUT/CLB netlist level (Figure 2-26c). Once again, such netlists are typically provided in encrypted form, either as encrypted EDIF or using some FPGA vendor-specific format. The reason for having placed-and-routed representations is to obtain the highest levels of performance. In some cases the placements will be relative, which means that the locations of all of the LUT, CLB, and other elements forming the block are fixed with respect to each other, but the block as a whole may be positioned anywhere (suitable) within the FPGA. Alternatively, in the case of IP blocks such as communications or bus protocol functions with specific I/O pin requirements, the placements of the elements forming the block may be absolute, which means that they cannot be changed in any way. Once again, the IP vendor may also provide a compiled cycle-accurate C/C++ model to be used for functional verification because such a model will simulate much faster than the LUT/CLB netlist-level model.

IP Core Generators

Another very common practice is for FPGA vendors (sometimes EDA vendors, IP providers, and even small, independent design houses) to provide special

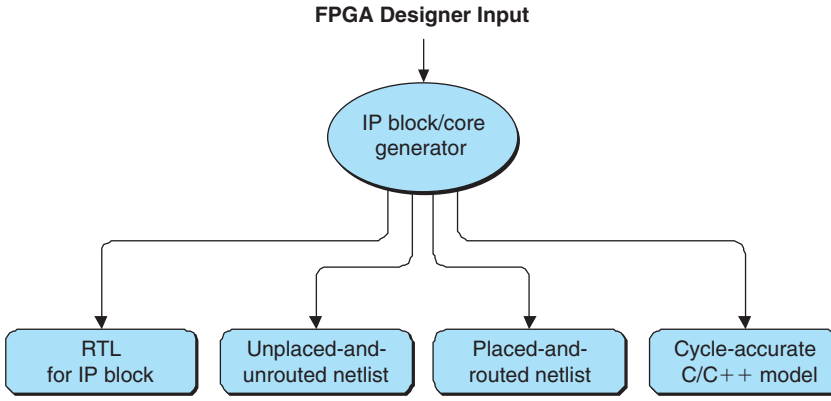


FIGURE 2-27 IP block/core generator.

tools that act as IP block/core generators. These generator applications are almost invariably parameterized, thereby allowing you to specify the widths and depths, or both of buses and functional elements.

First, you get to select from a list of different blocks/cores, and then you get to specify the parameters to be associated with each. Furthermore, in the case of some blocks/cores, the generator application may allow you to select from a list of functional elements that you wish to be included or excluded from the final representation. In the case of a communications block, for example, it might be possible to include or exclude certain error-checking logic. Or in the case of a CPU core, it might be possible to omit certain instructions or addressing modes. This allows the generator application to create the most efficient IP block/core in terms of its resource requirements and performance.

Depending on the origin of the generator application (or sometimes the licensing option you've signed up for), its output may be in the form of encrypted or unencrypted RTL source code, an unplaced-and-unrouted netlist, or a placed-and-routed netlist. In some cases, the generator may also output a cycle-accurate C/C++ model for use in simulation (Figure 2-27).

SYSTEM GATES VERSUS REAL GATES

One common metric used to measure the size of a device in the ASIC world is that of equivalent gates. The idea is that different vendors provide different functions in their cell libraries, where each implementation of each function requires a different number of transistors. This makes it difficult to compare the relative capacity and complexity of two devices.

The answer is to assign each function an equivalent gate value along the lines of "Function A equates to five equivalent gates; function B equates to

three equivalent gates…" The next step is to count all of the instances of each function, convert them into their equivalent gate values, sum all of these values together, and proudly proclaim, "My ASIC contains 10 million equivalent gates, which makes it much bigger than your ASIC!"

Unfortunately, nothing is simple because the definition of what actually constitutes an equivalent gate can vary depending on whom one is talking to. One common convention is for a 2-input NAND function to represent one equivalent gate. Alternatively, some vendors define an equivalent gate as equaling an arbitrary number of transistors. And a more esoteric convention defines an ECL equivalent gate as being "one-eleventh the minimum logic required to implement a single-bit full adder" (who on earth came up with this one?).

As usual, the best policy here is to make sure that everyone is talking about the same thing before releasing your grip on your hard-earned money.

And so we come to FPGAs. One of the problems FPGA vendors run into occurs when they are trying to establish a basis for comparison between their devices and ASICs. For example, if someone has an existing ASIC design that contains 500,000 equivalent gates and he wishes to migrate this design into an FPGA implementation, how can he tell if his design will fit into a particular FPGA? The fact that each 4-input LUT can be used to represent anywhere between one and more than twenty 2-input primitive logic gates makes such a comparison rather tricky.

In order to address this issue, FPGA vendors started talking about system gates in the early 1990s. Some folks say that this was a noble attempt to use terminology that ASIC designers could relate to, while others say that it was purely a marketing ploy that didn't do anyone any favors. Sad to relate, there appears to be no clear definition as to exactly what a system gate is. The situation was difficult enough when FPGAs essentially contained only generic programmable logic in the form of LUTs and registers. Even then, it was hard to state whether a particular ASIC design containing x equivalent gates could fit into an FPGA containing y system gates. This is because some ASIC designs may be predominantly combinatorial, while others may make excessively heavy use of registers. Both cases may result in a suboptimal mapping onto the FPGA.

The problem became worse when FPGAs started containing embedded blocks of RAM, because some functions can be implemented much more efficiently in RAM than in general-purpose logic. And the fact that LUTs can act as distributed RAM only serves to muddy the waters; for example, one vendor's system gate count values now include the qualifier, "Assumes 20 percent to 30 percent of LUTs are used as RAM." And, of course, the problems are exacerbated when we come to consider FPGAs containing embedded processor cores and similar functions, to the extent that some vendors now say, "System gate values are not meaningful for these devices."

Insider Info

Is there a rule of thumb that allows you to convert system gates to equivalent gates and vice versa? Sure, there are lots of them! Some folks say that if you are feeling optimistic, then you should divide the system gate value by three (in which case 3 million FPGA system gates would equate to 1 million ASIC equivalent gates, for example). Or if you're feeling a tad more on the pessimistic side, you could divide the system gates by five (in which case 3 million system gates would equate to 600,000 equivalent gates). However, other folks would say that the above is only true if you assume that the system gate's value encompasses all of the functions that you can implement using both the general-purpose programmable logic and the block RAMs. These folks would go on to say that if you remove the block RAMs from the equation, then you should divide the system gates value by ten (in which case, 3 million system gates would equate to only 300,000 equivalent gates), but in this case you still have the block RAMs to play with…arrggghhhh!

Ultimately, this topic spirals down into such a quagmire that even the FPGA vendors are trying desperately not to talk about system gates any more. When FPGAs were new on the scene, people were comfortable with the thought of equivalent gates and not so at ease considering designs in terms of LUTs, slices, and the like; however, the vast number of FPGA designs that have been undertaken over the years means that engineers are now much happier thinking in FPGA terms. For this reason, speaking as someone living in the trenches, I would prefer to see FPGAs specified and compared using only simple counts of:

Number of logic cells or logic elements or whatever (which equates to the number of 4-input LUTs and associated flip-flops/latches)
Number (and size) of embedded RAM blocks
Number (and size) of embedded multipliers
Number (and size) of embedded adders
Number (and size) of embedded MACs
etc.

INSTANT SUMMARY

Table 2-2 summarizes the key points associated with the various programming technologies discussed in this chapter.

TABLE 2-2 Summary of Programming Technologies

Feature	SRAM	Antifuse	E2PROM/FLASH
Technology node	State-of-the-art	One or more generations behind	One or more generations behind
Reprogrammable	Yes (in system)	No	Yes (in-system or offline)
Reprogramming speed (inc. erasing)	Fast	—	3x slower than SRAM
Volatile (must be programmed on power-up)	Yes	No	No (but can be if required)
Requires external configuration file	Yes	No	No
Good for prototyping	Yes (very good)	No	Yes (reasonable)
Instant-on	No	Yes	Yes
IP Security	Acceptable (especially when using bitstream encryption	Very good	Very good
Size of configuration cell	Large (six transistors)	Very small	Medium-small (two transistors)
Power consumption	Medium	Low	Medium
Rad Hard	No	Yes	Not really

Table 2-3 summarizes the key FPGA architectural features and choices available to designers.

TABLE 2-3 FPGA Architectural Features

Granularity	Fine, Medium, Coarse Grained
Logic Blocks	MUX-based, LUT-based
Embedded RAM	
Embedded Multipliers, Adders	
Embedded Processor Cores	
Clock Managers	
General-purpose I/O	
Gigabit Transceivers	
Intellectual Property	

Chapter 3

Programming (Configuring) an FPGA

In an Instant

Configuration Cells
Antifuse-based FPGAs
SRAM-based FPGAs
- Programming Embedded (Block) RAMs, Distributed RAMs, etc.
- Multiple Programming Chains
- Quickly Reinitializing the Device

Using the Configuration Port
- Serial Load with FPGA as Master
- Parallel Load with FPGA as Master
- Parallel Load with FPGA as Slave
- Serial Load with FPGA as Slave

Using the JTAG Port
Using an Embedded Processor
Instant Summary

Definitions

- *Configuration files* (also called *bit files*) contain the information that will be uploaded into the FPGA in order to program it to perform a specific function.
- In the case of SRAM-based FPGAs, the configuration file contains a mixture of *configuration data* (bits that are used to define the state of programmable logic elements directly) and *configuration commands* (instructions that tell the device what to do with the configuration data). When the configuration file is in the process of being loaded into the device, the information being transferred is referred to as the *configuration bitstream*.

—Technology Trade-offs—

- E^2-based and FLASH-based devices are programmed in a similar manner to their SRAM-based cousins. By comparison, in the case of antifuse-based FPGAs, the configuration file predominantly contains only a representation of the configuration data that will be used to grow the antifuses.

CONFIGURATION CELLS

The underlying concept associated with programming an FPGA is relatively simple (i.e., load the configuration file into the device). It can, however, be a little tricky to wrap one's brain around all of the different facets associated with this process, so we'll start with the basics and work our way up. Initially, let's assume we have a rudimentary device consisting only of an array of very simple programmable logic blocks surrounded by programmable interconnect (Figure 3-1).

Any facets of the device that may be programmed are done so by means of special configuration cells. The majority of FPGAs are based on the use of SRAM cells, but some employ FLASH (or E^2) cells, while others use antifuses.

Irrespective of the underlying technology, the device's interconnect has a large number of associated cells that can be used to configure it so as to connect the device's primary inputs and outputs to the programmable logic blocks and these logic blocks to each other. (In the case of the device's primary I/Os, which are not shown in Figure 3-1, each has a number of associated cells that can be used to configure them to accommodate specific I/O interface standards and so forth.)

For the purpose of this portion of our discussions, we shall assume that each programmable logic block comprises only a 4-input LUT, a multiplexer, and a register (Figure 3-2). The multiplexer requires an associated configuration cell to specify which input is to be selected. The register requires associated cells to specify whether it is to act as an edge-triggered flip-flop (as shown in Figure 3-2) or a level-sensitive latch, whether it is to be triggered by a positive- or negative-going clock edge (in the case of the flip-flop option) or an active-low or active-high enable (if the register is instructed to act as a latch), and whether it is to be initialized with a logic 0 or a logic 1. Meanwhile, the 4-input LUT is itself based on 16 configuration cells.

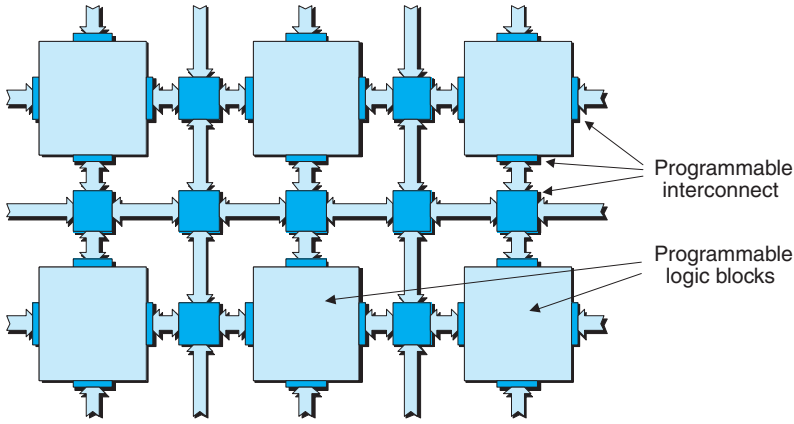


FIGURE 3-1 Top-down view of simple FPGA architecture.

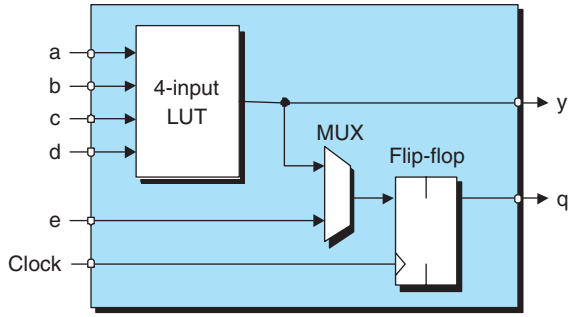


FIGURE 3-2 A very simple programmable logic block.

ANTIFUSE-BASED FPGAs

In the case of antifuse-based FPGAs, the antifuse cells can be visualized as scattered across the face of the device at strategic locations. The device is placed in a special device programmer, the configuration (bit) file is uploaded into the device programmer from the host computer, and the device programmer uses this file to guide it in applying pulses of relatively high voltage and current to selected pins to grow each antifuse in turn.

A very simplified way of thinking about this is that each antifuse has a "virtual" x-y location on the surface of the chip, where these x-y values are specified as integers. Based on this scenario, we can visualize using one group of I/O pins to represent the x value associated with a particular antifuse and another group of pins to represent the y value.

Once all of the fuses have been grown, the FPGA is removed from the device programmer and attached to a circuit board. Antifuse-based devices are, of course, one-time programmable (OTP) because once you've started the programming process, you're committed and it's too late to change your mind.

—Technology Trade-offs—

- Unlike SRAM-based FPGAs, FLASH-based devices are nonvolatile. They retain their configuration when power is removed from the system, and they don't need to be reprogrammed when power is reapplied to the system (although they can be if required).
- Also, FLASH-based devices can be programmed in-system (on the circuit board) or outside the system by means of a device programmer.

SRAM-BASED FPGAs

For the remainder of this chapter we shall consider only SRAM-based FPGAs. Remember that these devices are volatile, which means they have to be programmed in-system (on the circuit board), and they always need to be reprogrammed when power is first applied to the system.

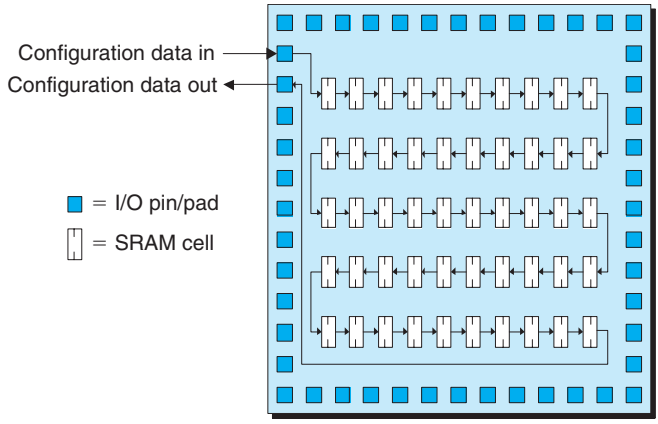


FIGURE 3-3 Visualizing the SRAM cells as a long shift register.

From the outside world, we can visualize all of the SRAM configuration cells as comprising a single (long) shift register. Consider a simple bird's-eye view of the surface of the chip showing only the I/O pins/pads and the SRAM configuration cells (Figure 3-3).

As a starting point, we shall assume that the beginning and end of this register chain are directly accessible from the outside world. However, it's important to note that this is only the case when using the *configuration port* programming mechanism in conjunction with the *serial load with FPGA as master* or *serial load with FPGA as slave* programming modes, as discussed below.

Also note that the *configuration data out* pin/signal shown in Figure 3-3 is only used if multiple FPGAs are to be configured by cascading (daisy-chaining) them together or if it is required to be able to read the configuration data back out of the device for any reason.

Insider Info

Programming an FPGA can take a significant amount of time. Consider a reasonably high-end device containing 25 million SRAM-based configuration cells. Programming such a device using a serial mode and a 25 MHz clock would take one second. This isn't too bad when you are first powering up a system, but it means that you really don't want to keep on reconfiguring the FPGA when the system is in operation.

Programming Embedded (Block) RAMs, Distributed RAMs, etc.

In the case of FPGAs containing large blocks of embedded (block) RAM, the cores of these blocks are implemented out of SRAM latches, and each of these latches is a configuration cell that forms a part of our "imaginary" register chain.

One interesting point is that each 4-input LUT (see Figure 3-2) can be configured to act as a LUT, as a small (16 × 1) chunk of distributed RAM, or as a 16-bit shift register. All of these manifestations employ the same group of 16 SRAM latches, where each of these latches is a configuration cell that forms a part of our imaginary register chain.

FAQ

How is the 16-bit shift register implemented?

A trick circuit is employed using the concept of a capacitive latch that prevents classic race conditions (this is pretty much the same way designers built flip-flops out of discrete transistors, resistors, and capacitors in the early 1960s).

Multiple Programming Chains

Figure 3-3 shows the configuration cells presented as a single programming chain. As there can be tens of millions of configuration cells, this chain can be very long indeed. Some FPGAs are architected so that the configuration port actually drives a number of smaller chains. This allows individual portions of the device to be configured and facilitates a variety of concepts such as modular and incremental design.

Quickly Reinitializing the Device

As was previously noted, the register in the programmable logic block has an associated configuration cell that specifies whether it is to be initialized with a logic 0 or a logic 1. Each FPGA family typically provides some mechanism such as an initialization pin that, when placed in its active state, causes all of these registers to be returned to their initialization values (this mechanism does not reinitialize any embedded [block] or distributed RAMs).

USING THE CONFIGURATION PORT

The early FPGAs made use of something called the *configuration port*. Even today, when more sophisticated techniques are available (like the JTAG interface discussed later in this chapter), this method is still widely used because it's relatively simple and is well understood by stalwarts in the FPGA fraternity.

We start with a small group of dedicated configuration mode pins that are used to inform the device which configuration mode is going to be used. In the early days, only two pins were employed to provide four modes.

Note that the names of the modes shown in this table—and also the relationship between the codes on the mode pins and the modes themselves—are

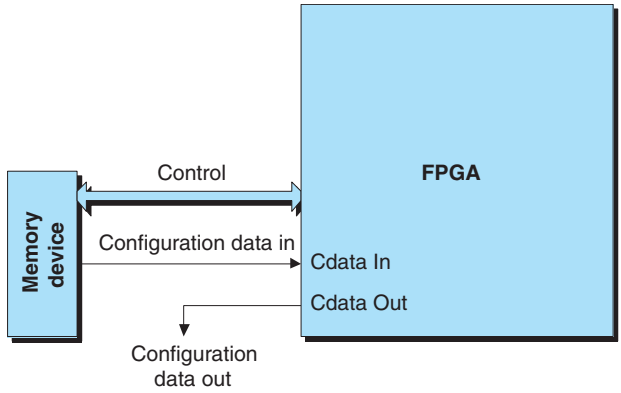


FIGURE 3-4 Serial load with FPGA as master.

intended for use only as an example. The actual codes and mode names vary from vendor to vendor.

The mode pins are typically hardwired to the desired logic 0 and logic 1 values at the circuit board level. (These pins can be driven from some other logic that allows the programming mode to be modified, but this is rarely done in practice.)

In addition to the hard-wired mode pins, an additional pin is used to instruct the FPGA to actually commence the configuration, while yet another pin is used by the device to report back when it's finished (there are also ways to determine if an error occurred during the process). This means that in addition to configuring the FPGA when the system is first powered up, the device may also be reinitialized using the original configuration data, if such an occurrence is deemed necessary.

The configuration port also makes use of additional pins to control the loading of the data and to input the data itself. The number of these pins depends on the configuration mode selected, as discussed below. The important point here is that once the configuration has been completed, most of these pins can subsequently be used as general-purpose I/O pins (we will return to this point a little later).

Serial Load with FPGA as Master

This is perhaps the simplest programming mode. In the early days, it involved the use of an external PROM. This was subsequently superceded by an EPROM, then an E^2PROM, and now—most commonly—a FLASH-based device. This special-purpose memory component has a single data output pin that is connected to a configuration data in pin on the FPGA (Figure 3-4).

The FPGA also uses several bits to control the external memory device, such as a reset signal to inform it when the FPGA is ready to start reading data and a clock signal to clock the data out.

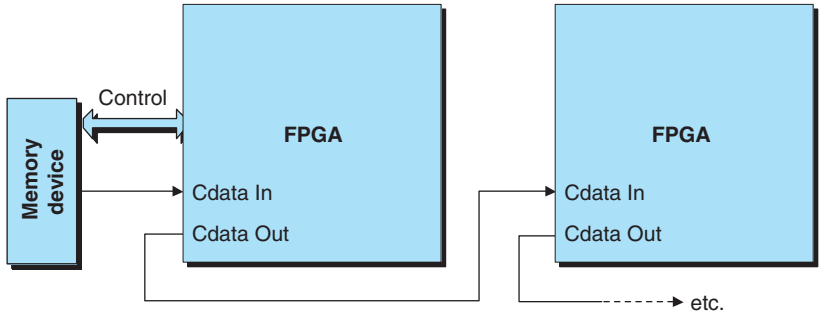


FIGURE 3-5 Daisy-chaining FPGAs.

The idea with this mode is that the FPGA doesn't need to supply the external memory device with a series of addresses. Instead, it simply pulses the reset signal to indicate that it wishes to start reading data from the beginning, and then it sends a series of clock pulses to clock the configuration data out of the memory device.

The configuration data out signal coming from the FPGA need only be connected if it is required to read the configuration data from the device for any reason. One such scenario occurs when there are multiple FPGAs on the circuit board. In this case, each could have its own dedicated external memory device and be configured in isolation, as shown in Figure 3-4. Alternatively, the FPGAs could be cascaded (daisy-chained) together and share a single external memory (Figure 3-5).

In this scenario, the first FPGA in the chain (the one connected directly to the external memory) would be configured to use the serial master mode, while the others would be serial slaves, as discussed later in this chapter.

Parallel Load with FPGA as Master

In many respects, this is very similar to the previous mode, except that the data is read in 8-bit chunks from a memory device with eight output pins. Groups of eight bits are very common and are referred to as bytes. In addition to providing control signals, the original FPGAs supplied the external memory device with an address that was used to point to whichever byte of configuration data was to be loaded next (Figure 3-6).

The way this worked was that the FPGA had an internal counter that was used to generate the address for the external memory. (The original FPGAs had 24-bit counters, which allowed them to address 16 million bytes of data.) At the beginning of the configuration sequence, this counter would be initialized with zero. After the byte of data pointed to by the counter had been read, the counter would be incremented to point to the next byte of data. This process would continue until all of the configuration data had been loaded.

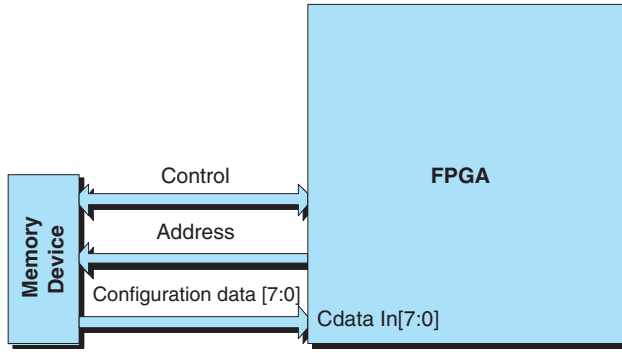


FIGURE 3-6 Parallel load with FPGA as master (original technique).

Insider Info

It's easy to assume that this parallel-loading technique offers speed advantages, but it didn't for quite some time. This is because—in early devices—as soon as a byte of data had been read into the device, it was clocked into the internal configuration shift register in a serial manner. Happily, this situation has been rectified in more modern FPGA families. On the other hand, although the eight pins can be used as general-purpose I/O pins once the configuration data has been loaded, in reality this is less than ideal. This is because these pins still have the tracks connecting them to the external memory device, which can cause a variety of signal integrity problems.

The real reason why this technique was so popular in the days of yore is that the special-purpose memory devices used in the serial load with FPGA as master mode were quite expensive. By comparison, this parallel technique allowed design engineers to use off-the-shelf memory devices, which were much cheaper.

Special-purpose memory devices created for use with FPGAs are now relatively inexpensive (and being FLASH-based, they are also reusable). Thus, modern FPGAs now use a new variation on this parallel-loading technique. In this case, the external memory is a special-purpose device that doesn't require an external address, which means that the FPGA no longer requires an internal counter for this purpose (Figure 3-7).

As for the serial mode discussed earlier, the FPGA simply pulses the external memory device's reset signal to indicate that it wishes to start reading data from the beginning, and then it sends a series of clock pulses to clock the configuration data out of the memory device.

Parallel Load with FPGA as Slave

The modes discussed above, in which the FPGA is the master, are attractive because of their inherent simplicity and because they only require the FPGA itself, along with a single external memory device.

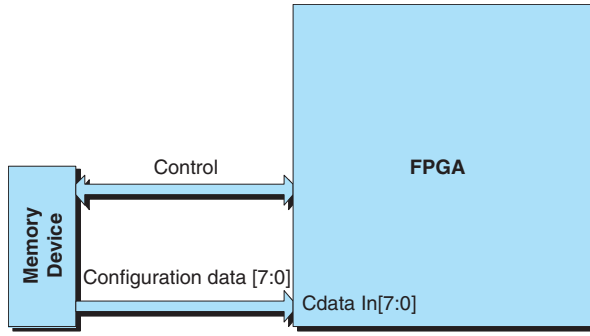


FIGURE 3-7 Parallel load with FPGA as the master (modern technique).

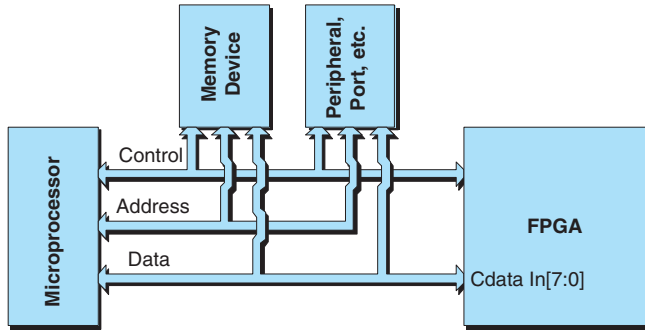


FIGURE 3-8 Parallel load with FPGA as slave.

However, a large number of circuit boards also include a microprocessor, which is typically already used to perform a wide variety of housekeeping tasks. In this case, the design engineers might decide to use the microprocessor to load the FPGA (Figure 3-8).

The idea here is that the microprocessor is in control. The microprocessor informs the FPGA when it wishes to commence the configuration process. It then reads a byte of data from the appropriate memory device (or peripheral, or whatever), writes that data into the FPGA, reads the next byte of data from the memory device, writes that byte into the FPGA, and so on until the configuration is complete.

This scenario conveys a number of advantages, not the least being that the microprocessor might be used to query the environment in which its surrounding system resides and to then select the configuration data to be loaded into the FPGA accordingly.

Serial Load with FPGA as Slave

This mode is almost identical to its parallel counterpart, except that only a single bit is used to load data into the FPGA (the microprocessor still reads data

out of the memory device one byte at a time, but it then converts this data into a series of bits to be written to the FPGA).

—Technology Trade-offs—

- The main advantage of this approach is that it uses fewer I/O pins on the FPGA. This means that—following the configuration process—only a single I/O pin has the additional track required to connect it to the microprocessor's data bus.

USING THE JTAG PORT

Like many other modern devices, today's FPGAs are equipped with a JTAG port. Standing for the *Joint Test Action Group* and officially known to engineers by its IEEE 1149.1 specification designator, JTAG was originally designed to implement the *boundary scan* technique for testing circuit boards and ICs.

A detailed description of JTAG and boundary scan is beyond the scope of this book. For our purposes here, it is sufficient to understand that the FPGA has a number of pins that are used as a JTAG port. One of these pins is used to input JTAG data, and another is used to output that data. Each of the FPGA's remaining I/O pins has an associated JTAG register (a flip-flop), where these registers are daisy-chained together (Figure 3-9).

The idea behind boundary scan is that, by means of the JTAG port, it's possible to serially clock data into the JTAG registers associated with the input pins, let the device (the FPGA in this case) operate on that data, store the results from this processing in the JTAG registers associated with the output pins, and, ultimately, to serially clock this result data back out of the JTAG port.

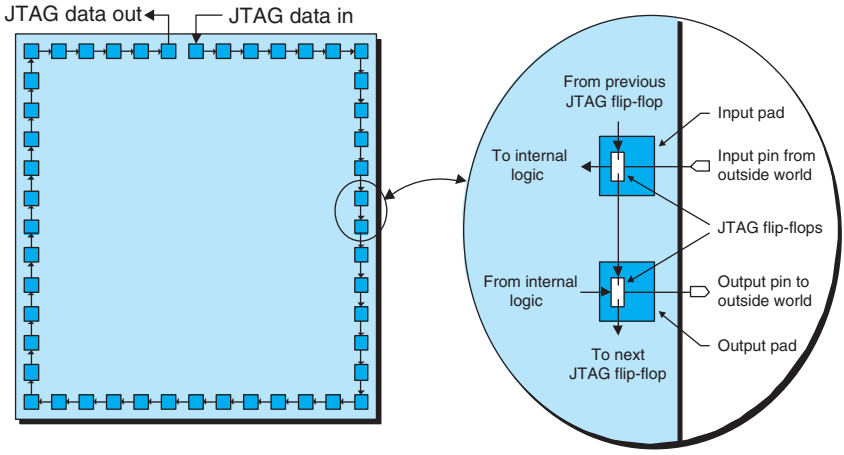


FIGURE 3-9 JTAG boundary scan registers.

However, JTAG devices also contain a variety of additional JTAG-related control logic, and, with regard to FPGAs, JTAG can be used for much more than boundary scans. For example, it's possible to issue special commands that are loaded into a special JTAG command register (not shown in Figure 3-9) by means of the JTAG port's data-in pin. One such command instructs the FPGA to connect its internal SRAM configuration shift register to the JTAG scan chain. In this case, the JTAG port can be used to program the FPGA. Thus, today's FPGAs now support five different programming modes and, therefore, require the use of three mode pins (additional modes may be added in the future).

Key Concept

Note that the JTAG port is always available, so the device can initially be configured via the traditional configuration port using one of the standard configuration modes, and it can subsequently be reconfigured using the JTAG port as required. Alternately, the device can be configured using only the JTAG port.

USING AN EMBEDDED PROCESSOR

But wait, there's more! We have discussed the fact that some FPGAs sport embedded processor cores, and each of these cores will have its own dedicated JTAG boundary scan chain. Consider an FPGA containing just one embedded processor (Figure 3-10).

The FPGA itself would only have a single external JTAG port. If required, a JTAG command can be loaded via this port that instructs the device to link the processor's local JTAG chain into the device's main JTAG chain. (Depending on the vendor, the two chains could be linked by default, in which case a complementary command could be used to disengage the internal chain.)

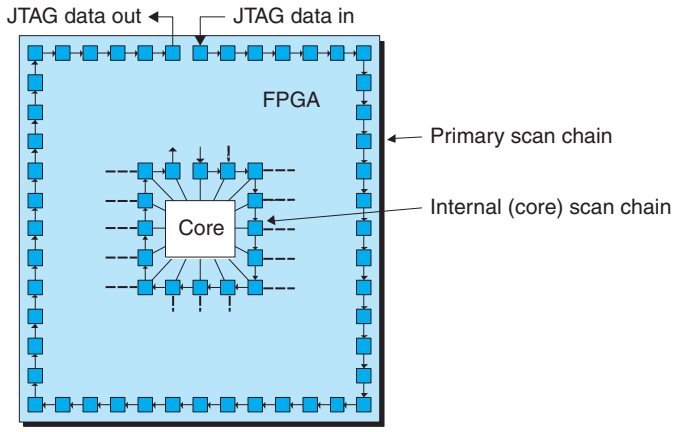


FIGURE 3-10 Embedded processor boundary scan chain.

The idea here is that the JTAG port can be used to initialize the internal microprocessor core (and associated peripherals) to the extent that the main body of the configuration process can then be handed over to the core. In some cases, the core might be used to query the environment in which the FPGA resides and to then select the configuration data to be loaded into the FPGA accordingly.

INSTANT SUMMARY

Table 3-1 shows the four original configuration modes, still widely used.

Table 3-2 summarizes today's five configuration modes.

TABLE 3-1 The Four Original Configuration Modes

Mode pins		Mode
0	0	Serial load with FPGA as master
0	1	Serial load with FPGA as slave
1	0	Parallel load with FPGA as master
1	1	Parallel load with FPGA as slave

TABLE 3-2 Today's Five Configuration Modes

Mode	pins		Mode
0	0	0	Serial load with FPGA as master
0	0	1	Serial load with FPGA as slave
0	1	0	Parallel load with FPGA as master
0	1	1	Parallel load with FPGA as slave
1	x	x	Use only the JTAG port

Chapter 4

FPGA vs. ASIC Designs

In an Instant

- When You Switch from ASIC to FPGA Design, or Vice Versa

Coding Styles

Pipelining and Levels of Logic

- Levels of Logic

Asynchronous Design Practices

- Asynchronous Structures
- Combinational Loops
- Delay Chains

Clock Considerations

- Clock Domains
- Clock Balancing
- Clock Gating versus Clock Enabling
- PLLs and Clock Conditioning Circuitry
- Reliable Data Transfer across Multiclock Domains

Register and Latch Considerations

- Latches
- Flip-flops with both "Set" and "Reset" Inputs
- Global Resets and Initial Conditions

Resource Sharing (Time-Division Multiplexing)

- Use It or Lose It!
- But Wait, There's More

State Machine Encoding

Test Methodologies

Migrating ASIC Designs to FPGAs and Vice Versa

- Alternative Design Scenarios

Instant Summary

Definitions

Here are some terms we'll be using in this chapter.

- *Pipelining* is analogous to an automobile assembly line, where the output of one logic block is the input of the next one. It will be explained in more detail in the following sections.
- *Latency* is the time in clock cycles that it takes for a specific block of data to work it way through a function, device, or system.
- *Levels of logic* refers to the number of gates between the inputs and outputs of a logic block.
- *Combinational loops* occur when the generation of a signal depends on itself feeding back through one or more logic gates.
- *Delay chains* are formed from a series of buffer or inverter gates and used for a variety of purposes, which we'll discuss later in this chapter.

When You Switch from ASIC to FPGA Design, or Vice Versa

Some design engineers have spent the best years of their young lives developing a seemingly endless series of ASICs, while others have languished in their cubicles learning the arcane secrets that are the province of the FPGA maestro. The problem arises when an engineer steeped in one of these implementation technologies is suddenly thrust into the antipodal domain. For example, a common scenario these days is for engineers who bask in the knowledge that they know everything there is to know about ASICs to be suddenly tasked with creating a design targeted toward an FPGA implementation.

This is a tricky topic because there are so many facets to it; the best we can hope for here is to provide an overview as to some of the more significant differences between ASIC and FPGA design styles.

CODING STYLES

When it comes to language-driven design flows, ASIC designers tend to write very portable code (in VHDL or Verilog) and to make the minimum use of instantiated (specifically named) cells.

By comparison, FPGA designers are more likely to instantiate specific low-level cells. For example, FPGA users may not be happy with the way the synthesis tool generates something like a multiplexer, so they may handcraft their own version and then instantiate it from within their code. Furthermore, pure FPGA users tend to use far more technology-specific attributes with regard to their synthesis engine than do their ASIC counterparts.

PIPELINING AND LEVELS OF LOGIC

FAQ

What is pipelining?

One tends to hear the word *pipelining* quite a lot, but this term is rarely explained in a clear way. Pipelining can be compared to an assembly line used in manufacturing automobiles. Assume that different specialists are needed for each step of the process: someone to attach the wheels to the chassis, the engine to the chassis, the body to the chassis, and paint the whole thing. It would be highly inefficient and time-consuming for all these specialists to sit around waiting for their turn to do their job. Instead, several cars are put on the assembly line at once, and each specialist does his or her job as the car moves down the line. Once the assembly line is in full flow, everyone will be working all the time and cars are created much more quickly.

We can often replicate this scenario in electronics. Assume we have a design (or a function forming part of a design) that can be implemented as a series of blocks of combinatorial logic (Figure 4-1). Let's say that each block takes

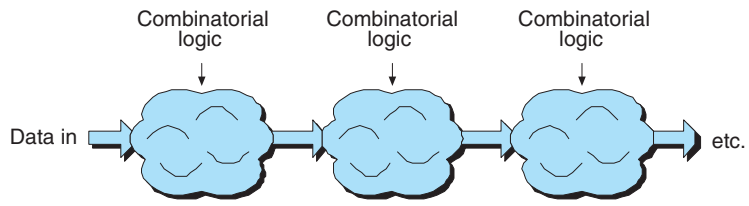


FIGURE 4-1 A function implemented using only combinatorial logic.

Y nanoseconds to perform its task and that we have five such blocks (of which only three are shown in Figure 4-1, of course). In this case, it will take 5 × Y nanoseconds for a word of data to propagate through the function, starting with its arrival at the inputs to the first block and ending with its departure from the outputs of the last block. However, we can instead use a pipelined design technique in which "islands" of combinatorial logic are sandwiched between blocks of registers.

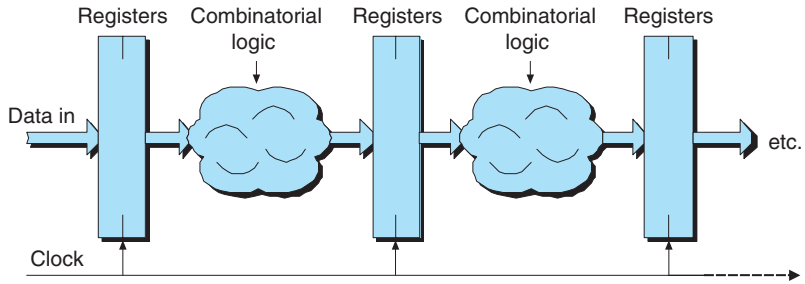


FIGURE 4-2 Pipelining the design.

Generally speaking, in the arrangement shown in Figure 4-1, we wouldn't want to present a new word of data to the inputs until we have stored the output results associated with the first word of data. This means that we end up with the same result as our inefficient car assembly scenario in that it takes a long time to process each word of data, and the majority of the workers (logic blocks) are sitting around twiddling their metaphorical thumbs for most of the time. In the pipelined design technique shown in Figure 4-2, all of the register banks are driven by a common clock signal. On each active clock edge, the registers feeding a block of logic are loaded with the results from the previous stage. These values then propagate through that block of logic until they arrive at its outputs, at which point they are ready to be loaded into the next set of registers on the next clock. In this case, as soon as "the pump has been primed" and the pipeline is fully loaded, a new word of data can be processed every Y nanoseconds.

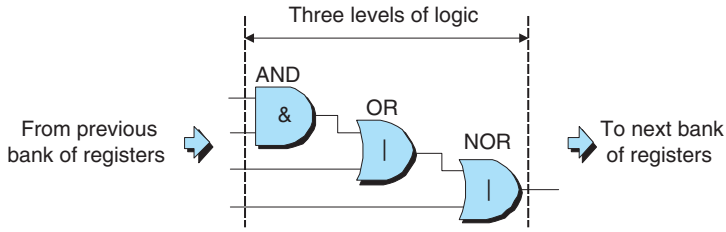


FIGURE 4-3 Levels of logic.

Levels of Logic

All of this boils down to the design engineer's having to perform a balancing act. Partitioning the combinational logic into smaller blocks and increasing the number of register stages will increase the performance of the design, but it will also consume more resources (and silicon real estate) on the chip and increase the latency of the design.

This is also the point where we start to run into the concept of levels of logic. For example, Figure 4-3 would be said to comprise three levels of logic because the worst-case path involves a signal having to pass through three gates before reaching the output.

In the case of an ASIC, a group of gates as shown in Figure 4-3 can be placed close to each other such that their track delays are very small. This means that, depending on the design, ASIC engineers can sometimes be a little sloppy about this sort of thing (it's not unheard of to have paths with, say, 15 or more levels of logic).

By comparison, if this sort of design were implemented on an FPGA with each of the gates implemented in a separate LUT, it would "fly like a brick" (go incredibly slowly) because the track delays on FPGAs are much more significant, relatively speaking. In reality, of course, a LUT can actually represent several levels of logic (the function shown in Figure 4-3 could be implemented in a single 4-input LUT), so the position isn't quite as dire as it may seem at first.

Having said this, the bottom line is that in order to bring up (or maintain) performance, FPGA designs tend to be more highly pipelined than their ASIC counterparts. This is facilitated by the fact that every FPGA logic cell tends to comprise both a LUT and a register, which makes registering the output very easy.

Key Concept

One way to think of latency is to return to the concept of an automobile assembly line. In this case, the throughput of the system might be one car rolling off the end of the line every minute. However, the latency of the system might be a full eight-hour shift since it takes hundreds of steps to finish a car (where each of these steps corresponds to a logic/register stage in a pipelined design).

ASYNCHRONOUS DESIGN PRACTICES

Asynchronous Structures

Depending on the task at hand, ASIC engineers may include asynchronous structures in their designs, where these constructs rely on the relative propagation delays of signals in order to function correctly. These techniques do not work in the FPGA world as the routing (and associated delays) can change dramatically with each new run of the place-and-route engines.

Combinational Loops

As a somewhat related topic, combinational loops are a major source of critical race conditions where logic values depend on routing delays. Although the practice is frowned upon in some circles, ASIC engineers can be little rapscallions when it comes to using these structures because they can fix track routing (and therefore the associated propagation delays) very precisely. This is not the case in the FPGA domain, so all such feedback loops should include a register element.

Delay Chains

Last but not least, ASIC engineers may use a series of buffer or inverter gates to create a delay chain. These delay chains may be used for a variety of purposes, such as addressing race conditions in asynchronous portions of the design. In addition to the delay from such a chain being hard to predict in the FPGA world, this type of structure increases the design's sensitivity to operating conditions, decreases its reliability, and can be a source of problems when migrating to another architecture or implementation technology.

CLOCK CONSIDERATIONS

Clock Domains

ASIC designs can feature a huge number of clocks (one hears of designs with more than 300 different clock domains). In the case of an FPGA, however, there are a limited number of dedicated global clock resources in any particular device. It is highly recommended that designers budget their clock systems to stay within the dedicated clock resources (as opposed to using general-purpose inputs as user-defined clocks).

Some FPGAs allow their clock trees to be fragmented into clock segments. If the target technology does support this feature, it should be identified and accounted for while mapping external or internal clocks.

Clock Balancing

In the case of ASIC designs, special techniques must be used to balance clock delays throughout the device. By comparison, FPGAs feature device-wide,

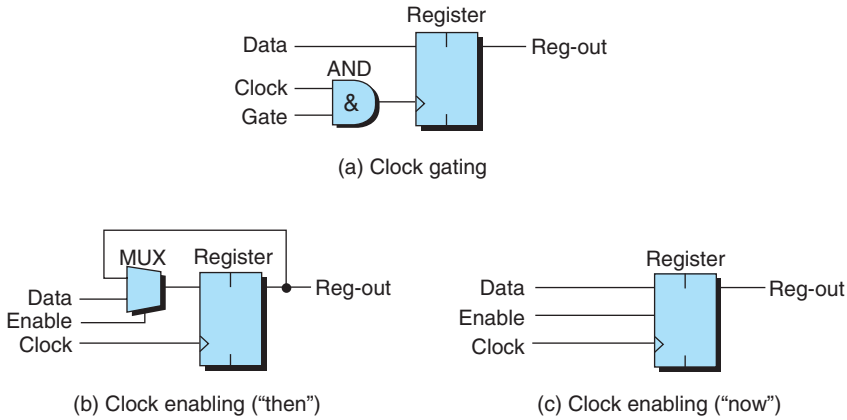


FIGURE 4-4 Clock gating versus clock enabling.

low-skew clock routing resources. This makes clock balancing unnecessary by the design engineer because the FPGA vendor has already taken care of it.

Clock Gating versus Clock Enabling

ASIC designs often use the technique of gated clocks to help reduce power dissipation, as shown in Figure 4-4a. However, these tend to give the design asynchronous characteristics and make it sensitive to glitches caused by inputs switching too closely together on the gating logic.

By comparison, FPGA designers tend to use the technique of enabling clocks. Originally this was performed by means of an external multiplexer as illustrated in Figure 4-4b; today, however, almost all FPGA architectures have a dedicated clock enable pin on the register itself, as shown in Figure 4-4c.

PLLs and Clock Conditioning Circuitry

FPGAs typically include PLL or DLL functions—one for each dedicated global clock (see also the discussions in Chapter 2). If these resources are used for on-chip clock generation, then the design should also include some mechanism for disabling or bypassing them to facilitate chip testing and debugging.

Reliable Data Transfer across Multiclock Domains

In reality, this topic is true for both ASIC and FPGA designs, the point being that the exchange of data between two independent clock domains must be performed very carefully to avoid losing or corrupting data. Bad synchronization may lead to metastability issues and tricky timing analysis problems. In order to achieve reliable transfers across domains, it is recommended to employ handshaking, double flopping, or asynchronous FIFO techniques.

REGISTER AND LATCH CONSIDERATIONS

Latches

ASIC engineers often make use of latches in their designs. As a general rule-of-thumb, if you are designing an FPGA, and you are tempted to use a latch, *don't*!

Flip-flops with both "Set" and "Reset" Inputs

Many ASIC libraries offer a wide range of flip-flops, including a selection that offer both set and reset inputs (both synchronous and asynchronous versions are usually available).

By comparison, FPGA flip-flops can usually be configured with either a set input or a reset input. In this case, implementing both set and reset inputs requires the use of a LUT, so FPGA design engineers often try to work around this and come up with an alternative implementation.

Global Resets and Initial Conditions

Every register in an FPGA is programmed with a default initial condition (that is, to contain a logic 0 or a logic 1).

Furthermore, the FPGA typically has a global reset signal that will return all of the registers (but not the embedded RAMs) to their initial conditions. ASIC designers typically don't implement anything equivalent to this capability.

RESOURCE SHARING (TIME-DIVISION MULTIPLEXING)

Resource sharing is an optimization technique that uses a single functional block (such as an adder or comparator) to implement several operations. For example, a multiplier may first be used to process two values called A and B, and then the same multiplier may be used to process two other values called C and D. (A good example of resource sharing is provided in Chapter 6.)

Another name for resource sharing is *time-division multiplexing* (TDM). Resources on an FPGA are more limited than on an ASIC. For this reason, FPGA designers tend to spend more effort on resource sharing than do their ASIC counterparts.

Use It or Lose It!

Actually, things are a little subtler than the brief note above might suggest because there is a fundamental use-it-or-lose-it consideration with regard to FPGA hardware. This means that FPGAs only come in certain sizes, so if you can't drop down to the next lower size, you might as well use everything that's available on the part you have.

For example, let's assume you have a design that requires two embedded hard processor cores. In addition to these processors, you might decide that by means of resource sharing, you could squeeze by with say 10 multipliers

and 2 megabytes of RAM. But if the only FPGA containing two processors also comes equipped with 50 multipliers and 10 megabytes of RAM, you can't get a refund, so you might as well make full use of the extra capabilities.

But Wait, There's More

In the case of FPGAs, getting data from LUTs/CLBs to and from special components like multipliers and MACs is usually more expensive (in terms of connectivity) than connecting with other LUTs/CLBs. Since resource sharing increases the amount of connectivity, you need to keep a watchful eye on this situation.

—Technology Trade-offs—

- In addition to the big components like multipliers and MACs, you can also share things like adders. Interestingly enough, in the carry-chain technologies (such as those fielded by Altera and Xilinx), as a first-order approximation, the cost of building an adder is pretty much equivalent to the cost of building a data bus's worth of sharing logic. For example, implementing two adders "as is" with completely independent inputs and outputs will cost you two adders and no resource-sharing multiplexers. But if you share, you will have one adder and two multiplexers (one for each set of inputs). In FPGA terms, this will be more expensive rather than less (in ASICs, the cost of a multiplexer is far less than the cost of an adder, so you would have a different trade-off point).

In the real world, the interactions between "using it or losing it" and connectivity costs are different for each technology and each situation; that is, Altera parts are different from Xilinx parts and so on.

STATE MACHINE ENCODING

The encoding scheme used for state machines is a good example of an area where what's good for an ASIC design might not be well suited for an FPGA implementation.

As we know, every LUT in an FPGA has a companion flip-flop. This usually means that there are a reasonable number of flip-flops sitting around waiting for something to do. In turn, this means that in many cases, a "one-hot" encoding scheme will be the best option for an FPGA-based state machine, especially if the activities in the various states are inherently independent.

Key Concept

The "one-hot" encoding scheme refers to the fact that each state in a state machine has its own state variable in the form of a flip-flop, and only one state variable may be active ("hot") at any particular time.

TEST METHODOLOGIES

ASIC designers typically spend a lot of time working with tools that perform SCAN chain insertion and *automatic test pattern generation* (ATPG). They may also include logic in their designs to perform *built-in self-test* (BIST). A large proportion of these efforts are intended to test the device for manufacturing defects. By comparison, FPGA designers typically don't worry about this form of device testing because FPGAs are preverified by the vendor.

Similarly, ASIC engineers typically expend a lot of effort inserting boundary scan (JTAG) into their designs and verifying them. By comparison, FPGAs already contain boundary scan capabilities in their fabric.

MIGRATING ASIC DESIGNS TO FPGAs AND VICE VERSA

Alternative Design Scenarios

When it comes to creating an FPGA design, there are a number of possible scenarios depending on what you are trying to do (Figure 4-5).

FPGA Only

This refers to a design that is intended for an FPGA implementation only. In this case, one might use any of the design flows and tools discussed elsewhere in this book.

FPGA-to-FPGA

This refers to taking an existing FPGA-based design and migrating it to a new FPGA technology (the new technology is often presented in the form of a new device family from the same FPGA vendor you used to implement the original design, but you may be moving to a new vendor also). With this scenario, it is rare that you will be performing a simple one-to-one migration, which means taking the contents of an existing component and migrating them directly

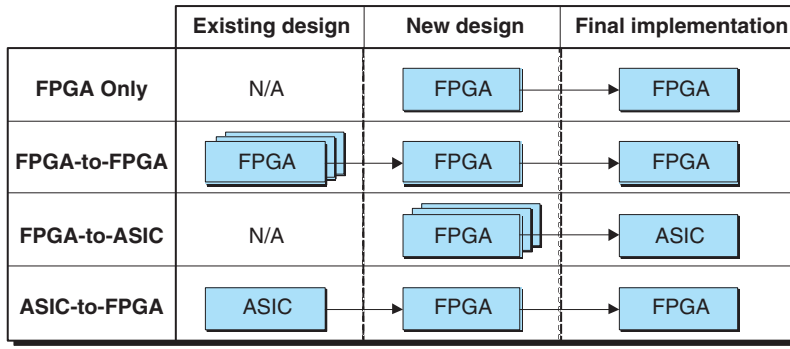


FIGURE 4-5 Alternative design scenarios.

to a new device. It is much more common to migrate the functionality from multiple existing FPGAs to a single new FPGA. Alternatively, you might be gathering the functionality of one or more existing FPGAs, plus a load of surrounding discrete logic, and bundling it all into a new device.

In these cases, the typical route is to gather all of the RTL code describing the original devices and discrete logic into a single design. The code may be tweaked to take advantage of any new features available in the targeted device and then resynthesized.

FPGA-to-ASIC

This refers to using one or more FPGAs to prototype an ASIC design. One big issue here is that, unless you're working with a small to medium ASIC, it is often necessary to partition the design across multiple FPGAs. Some EDA and FPGA vendors have (or used to have) applications that will perform this partitioning automatically, but tools like this come and go with the seasons. Also, their features and capabilities, along with the quality of their results, can change on an almost weekly basis (which is my roundabout way of telling you that you'll have to evaluate the latest offerings for yourself).

Another consideration is that functions like RAMs configured to act as FIFO memories or dual-port memories have specific realizations when they are implemented using embedded RAM blocks in FPGAs. These realizations are typically different from the way in which these functions will be implemented in an ASIC, which may cause problems. One solution is to create your own RTL library of ASIC functions for such things as multipliers, comparators, memory blocks, and the like that will give you a one-for-one mapping with their FPGA counterparts. Unfortunately, this means instantiating these elements in the RTL code for your design, as opposed to using generic RTL and letting the synthesis engine handle everything (so it's a balancing act like everything else in engineering).

As we discussed earlier, a design intended for an FPGA implementation typically contains fewer levels of logic between register stages than would a pure ASIC design. In some cases, it's best to create the RTL code associated with the design with the final ASIC implementation in mind and just take the hit with regard to reduced performance in the FPGA prototype.

Alternatively, one might generate two flavors of the RTL—one for use with the FPGA prototype and the other to provide the final ASIC. But this is generally regarded to be a horrible way to do things because it's easy for the two representations to lose synchronization and end up going in two totally different directions.

One way around this might be to use the pure C/C++ based tools introduced in Chapter 6. As you may recall, the idea here is that, as opposed to adding intelligence to the RTL source code by hand (thereby locking it into a target implementation), all of the intelligence is provided by your controlling and guiding the C/C++ synthesis engine itself (Figure 4-6).

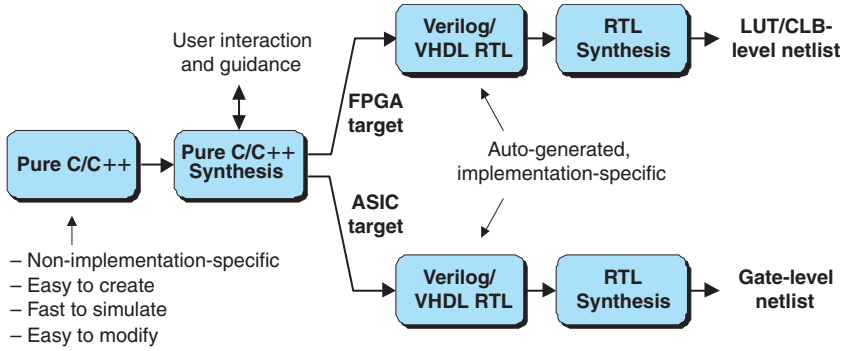


FIGURE 4-6 A pure C/C++-based design flow.

Once the synthesis engine has parsed the C/C++ source code, you can use it to perform microarchitecture trade-offs and evaluate their effects in terms of size and speed. The user-defined configuration associated with each "what-if" scenario can be named, saved, and reused as required. Thus, you could first create a configuration for use as an FPGA prototype and, once this had been verified, you could create a second configuration to be used for the final ASIC implementation. The key point is that the same C/C++ source code is used to drive both flows.

Another point to ponder is that a modern ASIC design can contain an unbelievable number of clock domains and subdomains (we're talking about hundreds of domains/subdomains here). By comparison, an FPGA has a limited number of primary clock domains (on the order of 10). This means that if you're using one or more FPGAs to prototype your ASIC, you're going to have to put a lot of thought into how you handle your clocks.

Insider Info

There's an interesting European Patent numbered EP0437491 (B1), which, when you read it—and, good grief, it's soooo boring—seems to lock down the idea of using multiple programmable devices like FPGAs to temporarily realize a design intended for final implementation as an ASIC. In reality, I think this patent was probably targeted toward using FPGAs to create a logic emulator, but the way it's worded would prevent anyone from using two or more FPGAs to prototype an ASIC.

ASIC-to-FPGA

This refers to taking an existing ASIC design and migrating it to an FPGA. The reasons for doing this are wide and varied, but they often involve the desire to tweak an existing ASIC's functionality without spending vast amounts of

money. Alternatively, the original ASIC technology may have become obsolete, but parts might still be required to support ongoing contracts (this is often the case with regard to military programs). One point of interest is that the latest generation of FPGAs has usually jumped so far so fast that it's possible to place an entire ASIC design from just a few years ago into a single modern FPGA (if you do have to partition the design across multiple FPGAs, then there are tools to aid you in this task, as discussed in the "FPGA-to-ASIC" section above). Here are the steps needed:

- First, you are going to have to go through your RTL code with a fine-tooth comb to **remove (or at least evaluate) any asynchronous logic, combinatorial loops, delay chains, and things of this ilk**. In the case of flip-flops with both set and reset inputs, you might wish to recode these to use only one or the other. You might also wish to look for any latches and redesign the circuit to use flip-flops instead.
- Also, you should keep a watchful eye open for **statements like if-then-else without the else clause** because, in these cases, synthesis tools will infer latches.
- In the case of clocks, you will have to **ensure that your target FPGA provides enough clock domains to handle the requirements of the original ASIC design**—otherwise, you'll have to redesign your clock circuitry.
- Furthermore, if your original ASIC design made use of **clock-gating techniques, you will have to strip these out and possibly replace them with clock-enable equivalents**. Once again, some FPGA and EDA vendors provide synthesis tools that can automatically convert an ASIC design using gated clocks to an equivalent FPGA design using clocks with enables.
- In the case of **complex functional elements such as memory blocks (e.g., FIFOs and dual-port RAMs), it will probably be necessary to tweak the RTL code to fit the design into the FPGA**. In some cases, this will involve replacing generic RTL statements (that will be processed by the synthesis engine) with calls to instantiate specific subcircuits or FPGA elements.
- Last, but not least, the **original pipelined ASIC design probably had more levels of logic between register elements than you would like in the FPGA implementation if you wish to maintain performance**. Most modern logic synthesis and physically aware tools provide retiming capability, which allows them to move logic back and forth across pipeline register boundaries to achieve better timing (the physically aware synthesis engines typically do a much better job at this; see also Chapter 7).
- It's also true that your modern FPGA is probably based on a later technology node (say, 130 nano) than your original ASIC design (say, 250 nano). This gives the FPGA an inherent speed advantage, which serves to offset its inherent track-delay disadvantages. At the end of the day, however, you may still end up **having to hand-tweak the code to add in more pipeline stages.**

INSTANT SUMMARY

Table 4-1 summarizes the design features of ASICs and FPGAs.

TABLE 4-1 Summary of Design Features of ASICs and FPGAs

	ASIC	FPGA
Coding styles	Portable code, minimal use of instantiated cells	Instantiate specific low level cells
Levels of logic	More levels of logic typically used	More highly pipelined
Asynchronous practices	May include asynchronous structures; can use delay chains	Do not include; no delay chains
Clock considerations	Large no. of clocks can be used; Special techniques needed to balance clock delays; use gated clock techniques	Limited no. of dedicated global clock resources; Onboard clock routing resources that make clock balancing unnecessary; Use enabling clocks
Register and latch considerations	Use latches	No latches
Global resets and initial conditions	No	Yes
Test methodologies	SCAN insertion; ATPG; BIST	FPGAs preverified by vendor; already contain boundary scan capabilities in fabric

Chapter 5

"Traditional" Design Flows

In an Instant

Schematic-based Design Flows
- Back-end Tools like Layout
- CAE + CAD = EDA
- A Simple (early) Schematic-driven ASIC Flow
- A Simple (early) Schematic-driven FPGA Flow
- Flat versus Hierarchical Schematics
- Schematic-driven FPGA Design Flows Today

HDL-based Design Flows
- Advent of HDL-based Flows
- A Plethora of HDLs
- Points to Ponder

Instant Summary

Definitions

Let's begin as usual by defining some terms we'll encounter in this chapter.

- *Schematic* is the common name for a circuit diagram.
- *Logic minimization* or *optimization* means replacing one group of gates with another that will perform the same task faster or use less real estate on the silicon.
- *Gate-level design* refers to a design represented as a collection of primitive logic gates and functions and the connections between them.
- *Electronic design automation* (EDA) is the name now applied to all of the CAE and CAD tools used to design electronic components and systems.
- *Hardware description languages* (HDLs) are computer languages used to describe hardware, namely the electronic portions of ICs and printed circuit boards.
- *Register transfer level* (RTL) is a higher level of abstraction than HDL. In RTL, the circuit is described as a collection of storage elements (registers), Boolean equations, control logic such as if-then-else statements, and complex sequences of events. The most popular languages used for capturing designs in RTL are VHDL and Verilog (with SystemVerilog starting to gain a larger following).

SCHEMATIC-BASED DESIGN FLOWS

First, let's briefly consider the way digital ICs were designed in the days of old—circa the early 1960s. The purpose of revisiting this ancient history is to establish an underlying framework that will facilitate understanding the more advanced design flows introduced in subsequent chapters.

In those days, electronic circuits were crafted by hand. Circuit diagrams—also known as *schematic diagrams* or just *schematics*—were hand-drawn and showed the symbols for the logic gates and functions that were to be used to implement the design, along with the connections between them. Each design team usually had at least one member who was really good at performing logic minimization and optimization. Checking that the design would work as planned insofar as its logical implementation—*functional verification*—was typically performed by a group of engineers sitting around a table working their way through the schematics saying, "Well, that looks OK." Similarly, timing verification—checking that the design met its required input-to-output and internal path delays and that no violation times (such as setup and hold parameters) associated with any of the internal registers were violated—was performed using a pencil and paper.

Insider Info

The wires connecting the logic gates on an integrated circuit may be referred to as wires, tracks, or interconnect, and all of these terms may be used interchangeably. In certain cases, the term metallization may also be used to refer to these tracks because they are predominantly formed by means of the IC's metal (metallization) layers.

Finally, a set of drawings representing the structures used to form the logic gates and the interconnections between them were drawn by hand. These drawings, which were formed from groups of simple polygons such as squares and rectangles, were subsequently used to create the photo-masks, which were themselves used to create the actual silicon chip.

Not surprisingly, this way of designing was time-consuming and prone to error. Something had to be done, and a number of companies and universities leapt into the fray in a variety of different directions. In the case of functional verification, for example, the late 1960s and early 1970s saw the advent of special programs in the form of rudimentary *logic simulators*.

In order to understand how these work, let's assume that we have a really simple gate-level design whose schematic diagram has been hand-drawn on paper (Figure 5-1). In order to use the logic simulator, the engineers first need to create a textual representation of the circuit called a gate-level netlist. In those days, the engineers would typically have been using a mainframe computer, and the netlist would have been captured as a set of punched cards called a deck.

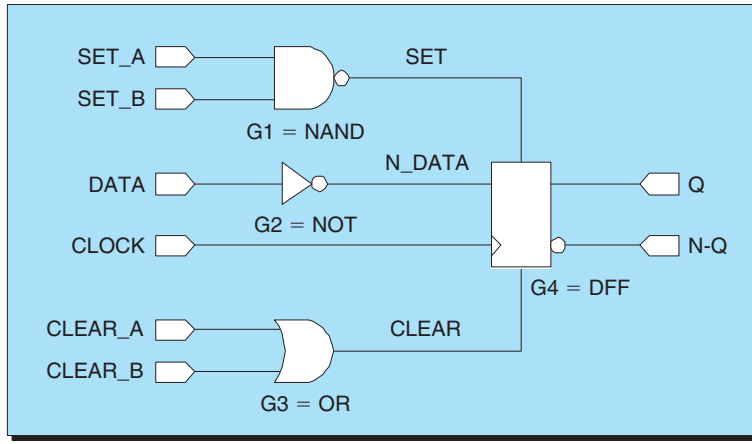


FIGURE 5-1 A simple schematic diagram (on paper).

```
BEGIN CIRCUIT=TEST
   INPUT  SET_A, SET-B, DATA, CLOCK, CLEAR_A, CLEAR_B;
   OUTPUT Q, N_Q;
   WIRE   SET, N_DATA, CLEAR;

   GATE G1=NAND (IN1=SET_A, IN2=SET_B, OUT1=SET);
   GATE G2=NOT  (IN1=DATA, OUT1=N_DATA);
   GATE G3=OR   (IN1=CLEAR_A, IN2=CLEAR_B, OUT1=CLEAR);
   GATE G4=DFF  (IN1=SET, IN2=N_DATA, IN3=CLOCK,
                 IN4=CLEAR, OUT1=Q, OUT2=N_Q);

END CIRCUIT=TEST;
```

FIGURE 5-2 A simple gate-level netlist (text file).

As computers (along with storage devices like hard disks) became more accessible, netlists began to be stored as text files (Figure 5-2).

It was also possible to associate delays with each logic gate. These delays—which are omitted here to keep things simple—were typically referenced as integer multiples of some core simulation time unit.

Note that the format shown in Figure 5-2 was made up purely for the purposes of this example. This was in keeping with the times because—just to keep everyone on their toes—anyone who created a tool like a logic simulator also tended to invent his or her own proprietary netlist language.

All of the early logic simulators had internal representations of primitive gates like AND, NAND, OR, NOR, etc. These were referred to as simulation primitives. Some simulators also had internal representations of more sophisticated functions like D-type flip-flops. In this case, the G4 = DFF function in Figure 5-2 would

```
                C C
                L L
      S S   C E E
      E E D L A A
      T T A O R R
        _ _ T C _ _
 TIME A B A K A B
----- -----------
    0 1 1 1 0 0 0    ; Set up initial values
  500 1 1 1 1 0 0    ; Rising edge on clock (load 0)
 1000 1 1 1 0 0 0    ; Falling edge on clock
 1500 1 1 0 0 0 0    ; Set data to 0 (N_data = 1)
 2000 1 1 0 1 0 0    ; Rising edge on clock (load 1)
 2500 1 1 0 1 0 1    ; Clear_B goes active (load 0)
   :
  etc.
```

FIGURE 5-3 A simple set of test vectors (text file).

map directly onto this internal representation. Alternatively, one could create a subcircuit called DFF, whose functionality was captured as a netlist of primitive AND, NAND, etc. gates. In this case, the G4 = DFF function in Figure 5-2 would actually be seen by the simulator as a call to instantiate a copy of this subcircuit.

Next, the user would create a set of *test vectors*—also known as *stimulus*—which were patterns of logic 0 and logic 1 values to be applied to the circuit's inputs. Once again, these test vectors were textual in nature, and they were typically presented in a tabular form looking something like that shown in Figure 5-3. The times at which the stimulus values were to be applied were shown in the left-hand column. The names of the input signals are presented vertically to save space.

As we know from Figures 5-1 and 5-2, there is an inverting (NOT) gate between the DATA input and the D-type flip-flop. Thus, when the DATA input is presented with 1 at time zero, this value will be inverted to a 0, which is the value that will be loaded into the register when the clock undergoes a rising (0-to-1) edge at time 500. Similarly, when the DATA input is presented with 0 at time 1,500, this value will be inverted to a 1, which is the value that will be loaded into the register when the clock undergoes its next rising (0-to-1) transition at time 2,000.

In today's terminology, the file of test vectors shown in Figure 5-3 would be considered a rudimentary testbench. Once again, time values were typically specified as integer multiples of some core simulation time unit.

The engineer would then invoke the logic simulator, which would read in the gate-level netlist and construct a virtual representation of the circuit in the computer's memory. The simulator would then read in the first test vector (the first line from the stimulus file), apply those values to the appropriate virtual inputs, and propagate their effects through the circuit. This would be repeated for each of the subsequent test vectors forming the testbench

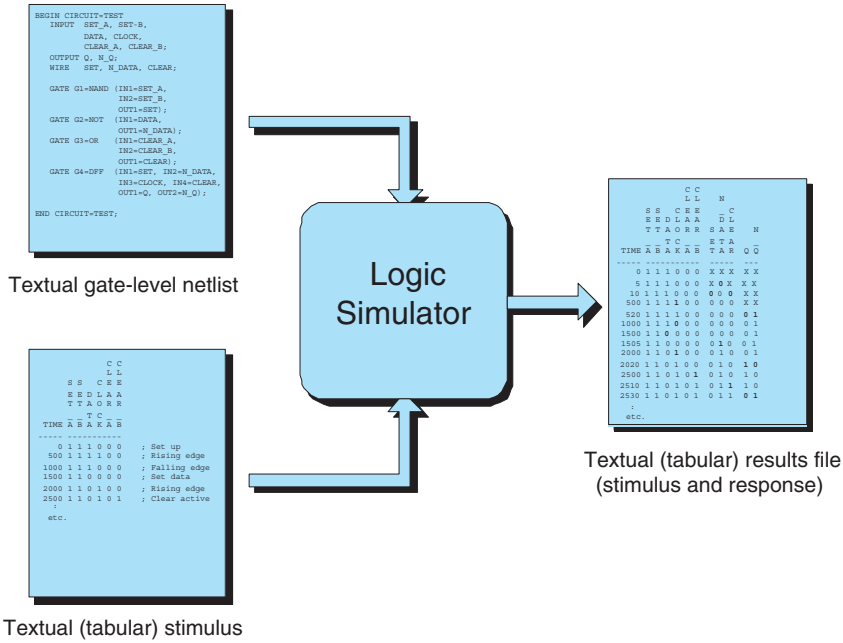


FIGURE 5-4 Running the logic simulator.

(Figure 5-4). The simulator would also use one or more control files (or online commands) to tell it which internal nodes (wires) and output pins to monitor, how long to simulate for, and so forth. The results, along with the original stimulus, would be stored in tabular form in a textual output file.

Let's assume that we've just traveled back in time and run one of the old simulators using the circuit represented in Figures 5-1 and 5-2 along with the stimulus shown in Figure 5-3. We will also assume that the NOT gate has a delay of five simulator time units associated with it, which means that a change on that gate's input will take five time units to propagate through the gate and appear on its output. Similarly, we'll assume that both the NAND and OR gates have associated delays of 10 time units, while the D-type flip-flop has associated delays of 20 time units.

In this case, if the simulator were instructed to monitor all of the internal nodes and output pins, the output file containing the simulation results would look something like that shown in Figure 5-5. For the purposes of our discussions, any changes to a signal's value are shown in bold font in this illustration, but this was not the case in the real world.

In this example, the initial values are applied to the input pins at time 0. At this time, all of the internal nodes and output pins show X values, which indicates unknown states. After five time units, the initial logic 1 that was applied to the DATA input propagates through the inverting NOT gate and appears as

```
                C C
                L L     N
      S S     C E E     _ C
      E E D L A A R     D L
      T T A O R R _ S A E     N
      _ _ T C _ _ A E T A     _
 TIME A B A K A B R T A R     Q Q
----- ----------- ----- -----
    0 1 1 1 0 0 0 X X X     X X  ; Set up initial values
    5 1 1 1 0 0 0 X 0 X     X X
   10 1 1 1 0 0 0 0 0 0     X X

  500 1 1 1 1 0 0 0 0 0     X X  ; Rising edge on clock
  520 1 1 1 1 0 0 0 0 0     0 1

 1000 1 1 1 0 0 0 0 0 0     0 1  ; Falling edge on clock

 1500 1 1 0 0 0 0 0 0 0     0 1  ; Set data to 0
 1505 1 1 0 0 0 0 0 1 0     0 1

 2000 1 1 0 1 0 0 0 1 0     0 1  ; Rising edge on clock
 2020 1 1 0 1 0 0 0 1 0     1 0

 2500 1 1 0 1 0 1 0 1 0     1 0  ; Clear_B goes active
 2510 1 1 0 1 0 1 0 1 1     1 0
 2530 1 1 0 1 0 1 0 1 1     0 1
   :
 etc.
```

FIGURE 5-5 Output results (text file).

a logic 0 on the internal N_DATA node. Similarly, at time 10, the initial values that were applied to the SET_A and SET_B inputs propagate through the NAND gate to the internal SET node, while the values on the CLEAR_A and CLEAR_B inputs propagate through the OR gate to the internal CLEAR node.

At time 500, a rising (0-to-1) edge on the CLOCK input causes the D-type flip-flop to load the value from the N_DATA node. The result appears on the Q and N_Q output pins 20 time units later. And so it goes.

Blank lines in the output file, such as the one shown between time 10 and time 500, were used to separate related groups of actions. For example, setting the initial values at time 0 caused signal changes at times 5 and 10. Then the transition on the CLOCK input at time 500 caused signal changes at time 520. As these two groups of actions were totally independent of each other, they were separated by a blank line.

It wasn't long before engineers were working with circuits that could contain thousands of gates and internal nodes along with simulation runs that could encompass thousands of time steps.

Back-end Tools like Layout

As opposed to tools like logic simulators that were intended to aid the engineers who were defining the function of ICs (and circuit boards), some companies focused on creating tools that would help in the process of laying the ICs out. In this context, *layout* refers to determining where to place the logic gates (actually, the transistors forming the logic gates) on the surface of the chip and how to route the wires between them.

These tools started out as early computer-aided drafting tools and evolved into interactive programs called polygon editors that allowed users to draw the polygons used to define the transistors and interconnect directly onto the computer screen. Descendants of these tools eventually gained the capability to accept the same netlist used to drive the logic simulator and to perform the layout (place-and-route) tasks automatically.

CAE + CAD = EDA

Tools like logic simulators that were used in the front-end (logical design capture and functional verification) portion of the design flow were originally gathered together under the umbrella name of *computer-aided engineering* (CAE). By comparison, tools like layout (*place-and-route*) that were used in the back-end (physical) portion of the design flow were originally gathered together under the name of *computer-aided design* (CAD).

Sometime during the 1980s, all of the CAE and CAD tools used to design electronic components and systems were gathered under the name *electronic design automation*, or EDA.

Insider Info

For historical reasons that are largely based on the origins of the terms CAE and CAD, the term design engineer—or simply engineer—typically refers to someone who works in the front-end of the design flow; that is, someone who performs tasks like conceiving and describing (capturing) the functionality of an IC (what it does and how it does it). By comparison, the term layout designer—or simply designer—commonly refers to someone who is ensconced in the back-end of the design flow; that is, someone who performs tasks such as laying out an IC (determining the locations of the gates and the routes of the tracks connecting them together).

A simple (early) Schematic-driven ASIC Flow

Toward the end of the 1970s and the beginning of the 1980s, some companies started providing graphical schematic capture programs that allowed engineers to create circuit (schematic) diagrams interactively. Using the mouse, an engineer could select symbols representing such entities as I/O pins and logic gates

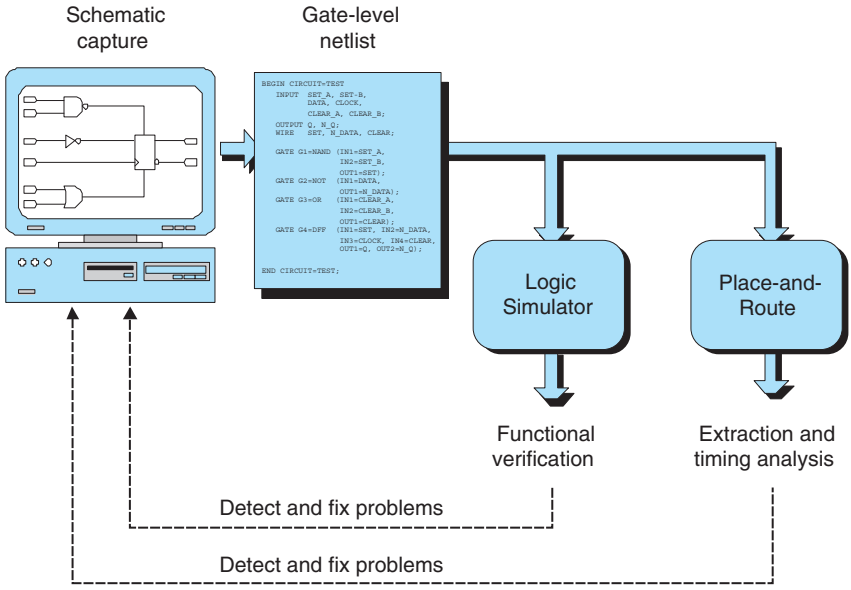


FIGURE 5-6 Simple (early) schematic-driven ASIC flow.

and functions from a special symbol library and place them on the screen. The engineer could then use the mouse to draw lines (wires) on the screen connecting the symbols together.

Once the circuit had been entered, the schematic capture package could be instructed to generate a corresponding gate-level netlist. This netlist could first be used to drive a logic simulator to verify the functionality of the design. The same netlist could then be used to drive the *place-and-route software* (Figure 5-6).

Any timing information that was initially used by the logic simulator would be estimated—particularly in the case of the tracks—and accurate timing analysis was only possible once all of the logic gates had been placed and the tracks connecting them had been routed. Thus, following place-and-route, an *extraction program* would be used to calculate the parasitic resistance and capacitance values associated with the structures (track segments, vias, transistors, etc.) forming the circuit. A *timing analysis program* would then use these values to generate a timing report for the device. In some flows, this timing information was also fed back to the logic simulator in order to perform a more accurate simulation.

—Technology Trade-offs—

- It's important to note here that, when creating the original schematic, the user would access the symbols for the logic gates and functions from a special

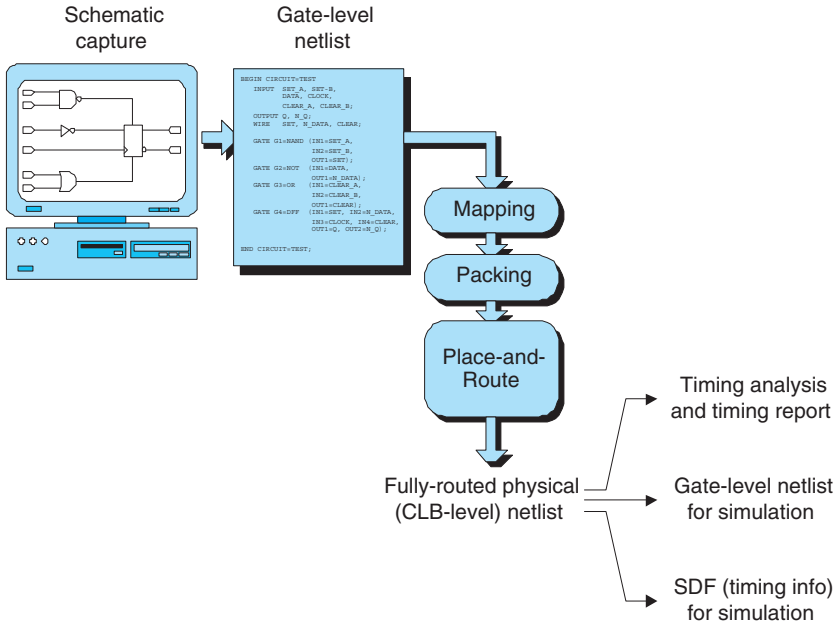


FIGURE 5-7 Simple (early) schematic-driven FPGA flow.

library that was associated with the targeted ASIC technology. Similarly, the simulator would be instructed to use a corresponding library of simulation models with the appropriate logical functionality and timing for the targeted ASIC technology. The result was that the gate-level netlist presented to the place-and-route software directly mapped onto the logic gates and functions being physically implemented on the silicon chip. This is a tad different from the FPGA flow, as discussed in the following subsection.

A Simple (early) Schematic-driven FPGA Flow

When the first FPGAs arrived on the scene in 1984, it was natural that their design flows would be based on existing schematic-driven ASIC flows. Indeed, the early portions of the flows were very similar in that, once again, a schematic capture package was used to represent the circuit as a collection of primitive logic gates and functions and to generate a corresponding gate-level netlist. As before, this netlist was subsequently used to drive the logic simulator to perform the functional verification.

The differences began with the implementation portion of the flow because the FPGA fabric consisted of an array of *configurable logic blocks* (CLBs), each of which was formed from a number of LUTs and registers. This required the introduction of some additional steps called *mapping* and *packing* into the flow (Figure 5-7).

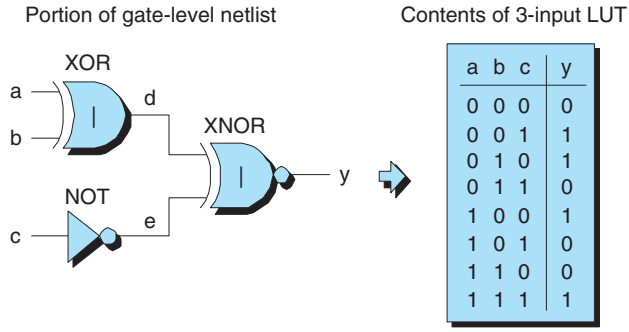


FIGURE 5-8 Mapping logic gates into LUTs.

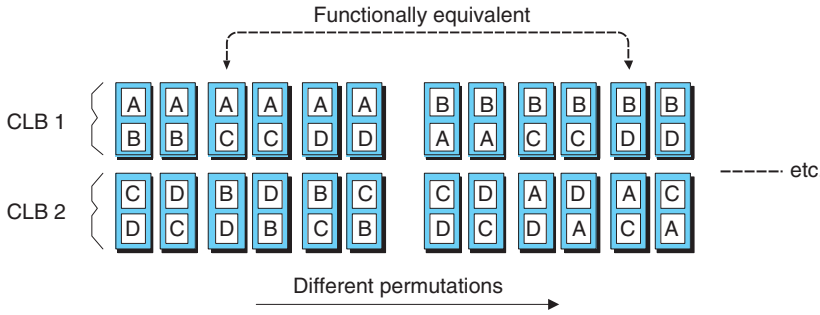


FIGURE 5-9 Packing LUTs into CLBs.

Mapping

In this context, *mapping* refers to the process of associating entities such as the gate-level functions in the gate-level netlist with the LUT-level functions available on the FPGA. Of course, this isn't a one-for-one mapping because each LUT can be used to represent a number of logic gates (Figure 5-8). Mapping (which is still performed today, but elsewhere in the flow, as will be discussed later) is a nontrivial problem because there are a large number of ways in which the logic gates forming a netlist can be partitioned into the smaller groups to be mapped into LUTs. As a simple example, the functionality of the NOT gate shown in Figure 5-8 might have been omitted from this LUT and instead incorporated into the upstream LUT driving wire c.

Packing

Only 12 of the 24 possible permutations are shown here. Furthermore, in reality there are actually only 12 permutations of significance because each has a "mirror image" that is functionally its equivalent, such as the AC-BD and BD-AC pairs shown in Figure 5-9. The reason for this is that when we come to place-and-route, the relative locations of the two CLBs can be exchanged.

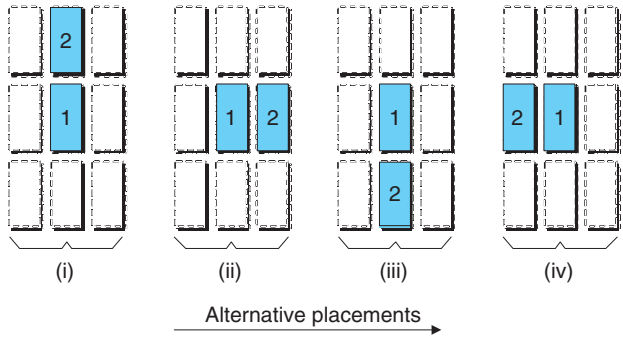


FIGURE 5-10 Placing the CLBs.

Place-and-route

Following packing, we move to *place-and-route*. With regard to the previous point, let's assume that our two CLBs need to be connected together, but that—purely for the purposes of this portion of our discussions—they can only be placed horizontally or vertically adjacent to each other, in which case there are four possibilities (Figure 5-10).

In the case of placement (i) for example, if CLB 1 contained LUTs A-C and CLB 2 contained LUTs B-D, this would be identical to swapping the positions of the two CLBs and exchanging their contents.

If we only had the two CLBs shown in Figure 5-10, it would be easy to determine their optimal placement with respect to each other (which would have to be one of the four options shown above) and the absolute placement of this two-CLB group with respect to the entire chip.

—*Technology Trade-offs*—

- The placement problem is much more complex in the real world because a real design can contain extremely large numbers of CLBs. In addition to CLBs 1 and 2 being connected together, they will almost certainly need to be connected to other CLBs. For example, CLB 1 may also need to be connected to CLBs 3, 5, and 8, while CLB 2 may need to be connected to CLBs 4, 6, 7, and 8. And each of these new CLBs may need to be connected to each other or to yet more CLBs. Thus, although placing CLBs 1 and 2 next to each other would be best for them, it might be detrimental to their relationships with the other CLBs, and the most optimal solution overall might be to separate CLBs 1 and 2 by some amount.

Although placement is difficult, deciding on the optimal way to route the signals between the various CLBs poses an even more Byzantine problem. The complexity of these tasks is mind-boggling, so we'll leave it to those guys and gals who write the place-and-route algorithms.

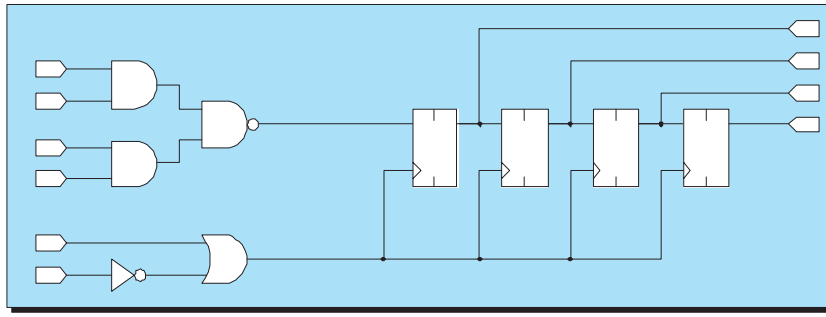


FIGURE 5-11 Simple schematic drawn on a piece of paper.

Timing Analysis and Post-place-and-route Simulation

Following place-and-route, we have a fully routed physical (CLB-level) netlist, as was illustrated in Figure 5-7. At this point, a *static timing analysis* (STA) utility will be run to calculate all of the input-to-output and internal path delays and to check for any timing violations (setup, hold, etc.) associated with any of the internal registers.

One interesting point occurs if the design engineers wish to resimulate their design with accurate (post-place-and-route) timing information. In this case, they have to use the FPGA tool suite to generate a new gate-level netlist along with associated timing information in the form of an industry-standard file format called—perhaps not surprisingly—*standard delay format* (SDF). The main reason for generating this new gate-level netlist is that—once the original netlist has been coerced into its CLB-level equivalent—it simply isn't possible to relate the timings associated with this new representation back into the original gate-level incarnation.

Flat versus Hierarchical Schematics

Clunky Flat Schematics

The very first schematic packages essentially allowed a design to be captured as a humongous, flat circuit diagram split into a number of "pages." You created a single flat schematic as a series of pages linked together by interpage connector symbols, where the names you gave these symbols told the system which ones were to be connected together. For example, consider a simple circuit sketched on a piece of paper (Figure 5-11).

Assume that the gates on the left represent some control logic, while the four registers on the right are implementing a 4-bit shift register. Obviously, this is a trivial example, and a real circuit would have many more logic gates. We're just trying to tie down some underlying concepts here, such as the fact that when you entered this circuit into your schematic capture system, you might split it into two pages (Figure 5-12).

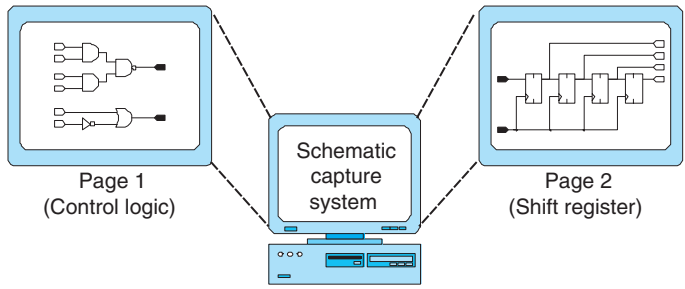


FIGURE 5-12 Simple two-page flat schematic.

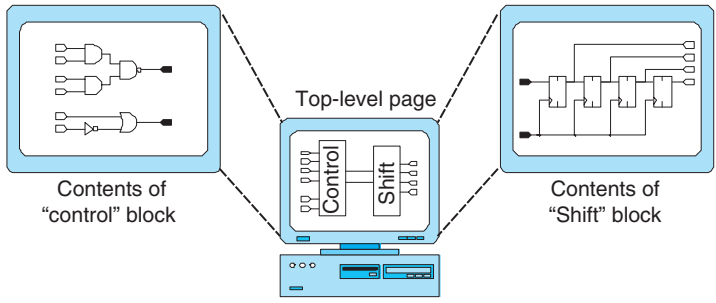


FIGURE 5-13 Simple hierarchical schematic.

Sleek Hierarchical (block-based) Schematics

There were a number of problems associated with the flat schematics, especially when dealing with real-world circuits requiring 50 or more pages:

- It was difficult to visualize a high-level, top-down view of the design.
- It was difficult to save and reuse portions of the design in future projects.
- In the case of designs in which some portion of the circuit was repeated multiple times (which is very common), that portion would have to be redrawn or copied onto multiple pages. This became very painful if you subsequently realized that you had to make a change because you would have to make the same change to all the copies.

The answer was to enhance schematic capture packages to support the concept of hierarchy. In the case of our shift register circuit, for example, you might start with a top-level page in which you would create two blocks called control and shift, each with the requisite number of input and output pins. You would then connect these blocks to each other and to some primary inputs and outputs.

Next, you would instruct the system to "push down" into the control block, which would open a new schematic page. If you were lucky, the system would automatically pre-populate this page with input and output connector symbols (and with associated names) corresponding to the pins on its parent block. You would then create the schematic corresponding to that block as usual (Figure 5-13).

In fact, each block could contain a further block-level schematic, or a gate-level schematic, or (very commonly) a mixture of both. These hierarchical block-based schematics answered the problems associated with flat schematics:

- They made it easier to visualize a high-level, top-down view of the design and to work one's way through the design.
- They made it easier to save and reuse portions of the design in future projects.
- In the case of designs in which some portion of the circuit was repeated multiple times, it was only necessary to create that portion—as a discrete block—once and then to instantiate (call) that block multiple times. This made things easy if you subsequently realized that you had to make a change because you would only have to modify the contents of the initial block.

Schematic-driven FPGA Design Flows Today

All of the original schematic, mapping, packing, and place-and-route applications were typically created and owned by the FPGA companies. However, the general feeling is that a company can either be good at creating EDA tools or it can be good at creating silicon chips, but not both.

Another facet of the problem is that design tools were originally extremely expensive in the ASIC world (even tools like schematic capture, which today are commonly regarded as commodity products). By comparison, the FPGA vendors were focused on selling chips, so right from the get-go they offered their tools at a very low cost (in fact, if you were a big enough customer, they'd give you the entire design tool suite for free). While this had its obvious attractions to the end user, the downside was that the FPGA vendors weren't too keen on spending vast amounts of money enhancing tools for which they received little recompense.

Over time, therefore, external EDA vendors started to supply portions of the puzzle, starting with schematic capture and then moving into mapping and packing. Having said this, the FPGA vendors still typically provide internally developed, less sophisticated (compared to the state-of-the-art) versions of tools like schematic capture as part of their basic tool suite, and they also maintain a Vulcan Death Grip on their crown jewels (the place-and-route software).

Insider Info

For many engineers today, driving a design using schematic capture at the gate-level of abstraction is but a distant memory. In some cases, FPGA vendors offer little support for this type of flow for their latest devices to the extent that they only provide schematic libraries for older component generations. However, schematic capture does still find a role with some older engineers and with folks who need to make minor functional changes to legacy designs. Furthermore, graphical entry mechanisms that are descended from early schematic capture packages still find a place in modern design flows.

HDL-BASED DESIGN FLOWS

Advent of HDL-based Flows

Toward the end of the 1980s, as designs grew in size and complexity, schematic-based ASIC flows began to run out of steam. Visualizing, capturing, debugging, understanding, and maintaining a design at the gate level of abstraction became increasingly difficult and inefficient when juggling 5,000 or more gates and reams of schematic pages. In addition to the fact that capturing a large design at the gate level of abstraction is prone to error, it is extremely time-consuming. Thus, some EDA vendors started to develop design tools and flows based on the use of *hardware description languages*, or HDLs.

The idea behind a hardware description language is, perhaps not surprisingly, that you can use it to describe hardware, in particular the electronic portions (components and wires) of ICs and printed circuit boards. (The HDL may also be used to provide limited representations of the cables and connectors linking circuit boards together.)

In the early days of electronics, almost anyone who created an EDA tool created his or her own HDL to go with it. Some of these were analog HDLs in that they were intended to represent circuits in the analog domain, while others were focused on representing digital functionality. For the purposes of this book, we are interested in HDLs only in the context of designing digital ICs in the form of ASICs and FPGAs.

Some of the more popular digital HDLs are introduced later in this chapter. For the nonce, however, let's focus more on how a generic digital HDL is used as part of a design flow. The first thing to note is that the functionality of a digital circuit can be represented at different levels of abstraction and that different HDLs support these levels of abstraction to a greater or lesser extent (Figure 5-14).

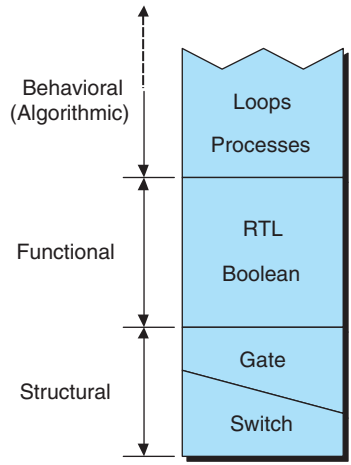


FIGURE 5-14 Different levels of abstraction.

The lowest level of abstraction for a digital HDL would be the **switch level**, which refers to the ability to describe the circuit as a netlist of transistor switches.

A slightly higher level of abstraction would be the **gate level**, which refers to the ability to describe the circuit as a netlist of primitive logic gates and functions. Thus, the early gate-level netlist formats generated by schematic capture packages as discussed in the previous section were in fact rudimentary HDLs.

ALERT!

Both switch-level and gate-level netlists may be classed as structural representations. It should be noted, however, that "structural" can have different connotations because it may also be used to refer to a hierarchical block-level netlist in which each block may have its contents specified using any of the levels of abstraction shown in Figure 5-14.

The next level of HDL sophistication is the ability to support **functional representations**, which covers a range of constructs. At the lower end is the capability to describe a function using Boolean equations. For example, assuming we had already declared a set of signals called Y, SELECT, DATA-A, and DATA-B, we could capture the functionality of a simple 2:1 multiplexer using the following Boolean equation:

```
Y = (SELECT & DATA-A)|(!SELECT & DATA-B);
```

Note that this is a generic syntax that does not favor any particular HDL and is used only for the purposes of this example.

The functional level of abstraction also encompasses register transfer level (RTL) representations. The term RTL covers a multitude of manifestations, but the easiest way to wrap one's brain around the underlying concept is to consider a design formed from a collection of registers linked by combinational logic. These registers are often controlled by a common clock signal, so assuming that we have already declared two signals called CLOCK and CONTROL, along with a set of registers called REGA, REGB, REGC, and REGD, then an RTL-type statement might look something like the following:

```
when CLOCK rises if CONTROL == "1"
then REGA = REGB & REGC; else REGA = REGB|REGD;
end if;
end when;
```

In this case, symbols like *when*, *rises*, *if*, *then*, *else*, and the like are keywords whose semantics are defined by the owners of the HDL. Once again, this is a generic syntax that does not favor any particular HDL and is used only for the purposes of this example.

The highest level of abstraction sported by traditional HDLs is known as **behavioral**, which refers to the ability to describe the behavior of a circuit using abstract constructs like loops and processes. This also encompasses using algorithmic elements like adders and multipliers in equations; for example:

$$Y = (DATA\text{-}A + DATA\text{-}B) * DATA\text{-}C;$$

We should note that there is also a system level of abstraction (not shown in Figure 5-14) that features constructs intended for system-level design applications, but we'll worry about this level a little later.

Many of the early digital HDLs supported only structural representations in the form of switch or gate-level netlists. Others such as ABEL, CUPL, and PALASM were used to capture the required functionality for PLD devices. These languages supported different levels of functional abstraction, such as Boolean equations, text-based truth tables, and text-based finite state machine (FSM) descriptions.

The next generation of HDLs, which were predominantly targeted toward logic simulation, supported more sophisticated levels of abstraction such as RTL and some behavioral constructs. It was these HDLs that formed the core of the first true HDL-based design flows.

A Simple (early) HDL-based ASIC Flow

The key feature of HDL-based ASIC design flows is their use of logic synthesis technology, which began to appear on the market around the mid-1980s. These tools could accept an RTL representation of a design along with a set of timing constraints. In this case, the timing constraints were presented in a side-file containing statements along the lines of "the maximum delay from input X to output Y should be no greater than N nanoseconds" (the actual format would be a little drier and more boring).

The logic synthesis application automatically converted the RTL representation into a mixture of registers and Boolean equations, performed a variety of minimizations and optimizations (including optimizing for area and timing), and then generated a gate-level netlist that would (or at least, should) meet the original timing constraints (Figure 5-15).

—Technology Trade-offs—

- There were a number of advantages to this new type of flow. First the productivity of the design engineers rose dramatically because it was much easier to specify, understand, discuss, and debug the required functionality of the design at the RTL level of abstraction as opposed to working with reams of gate-level schematics.
- Also, logic simulators could run designs described in RTL much more quickly than their gate-level counterparts.

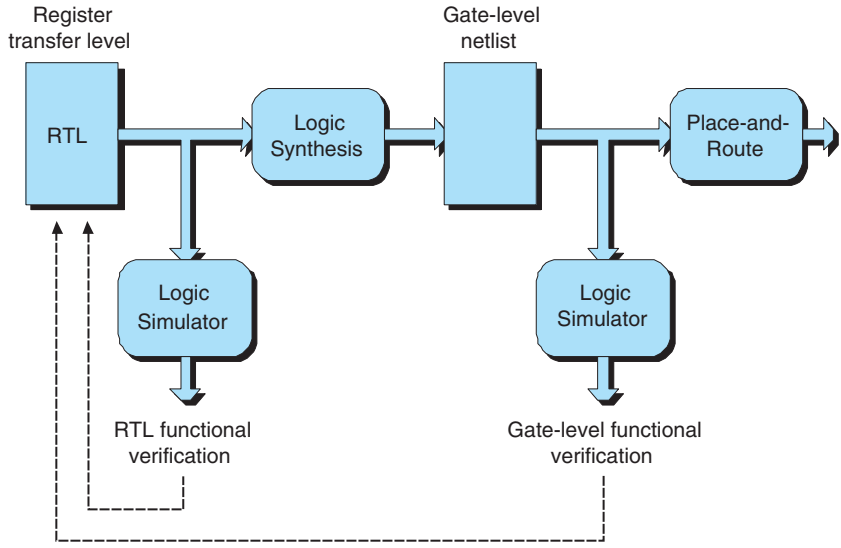


FIGURE 5-15 Simple HDL-based ASIC flow.

- One slight glitch was that logic simulators could work with designs specified at high levels of abstraction that included behavioral constructs, but early synthesis tools could only accept functional representations up to the level of RTL. Thus, design engineers were obliged to work with a synthesizable subset of their HDL of choice.

Once the synthesis tool had generated a gate-level netlist, the flow became very similar to the schematic-based ASIC flows discussed in the previous chapter. The gate-level netlist could be simulated to ensure its functional validity, and it could also be used to perform timing analysis based on estimated values for tracks and other circuit elements. The netlist could then be used to drive the place-and-route software, following which a more accurate timing analysis could be performed using extracted resistance and linefeed capacitance values.

A Simple (early) HDL-based FPGA Flow

It took some time for HDL-based flows to flourish within the ASIC community. Meanwhile, design engineers were still coming to grips with the concept of FPGAs. Thus, it wasn't until the very early 1990s that HDL-based flows featuring logic synthesis technology became fully available in the FPGA world (Figure 5-16).

As before, once the synthesis tool had generated a gate-level netlist, the flow became very similar to the schematic-based FPGA flows discussed in the previous chapter. The gate-level netlist could be simulated to ensure its functional validity, and it could also be used to perform timing analysis based on estimated values for tracks and other circuit elements. The netlist could then

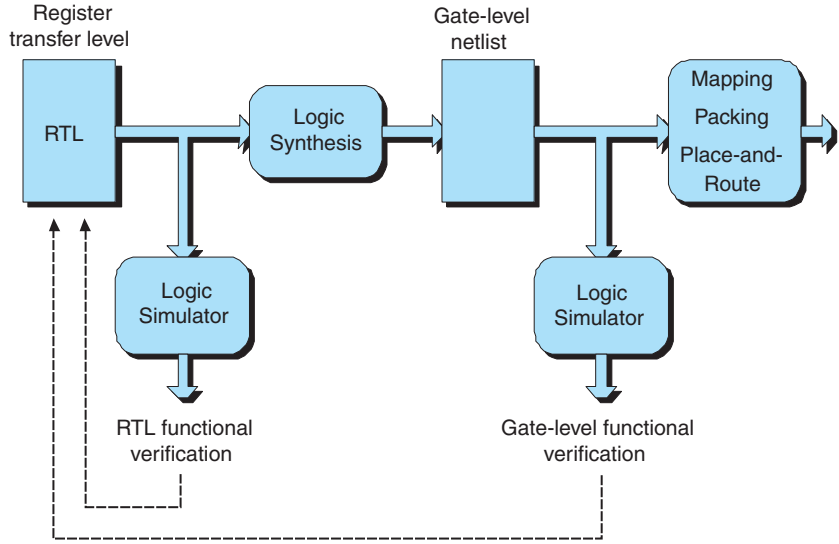


FIGURE 5-16 Simple HDL-based FPGA flow.

be used to drive the FPGA's mapping, packing, and place-and-route software, following which a more accurate timing report could be generated using real-world (physical) values.

Architecturally Aware FPGA Flows

The main problem besetting the original HDL-based FPGA flows was that their logic synthesis technologies were derived from the ASIC world. Thus, these tools "thought" in terms of primitive logic gates and registers. In turn, this meant that they output gate-level netlists, and it was left to the FPGA vendor to perform the mapping, packing, and place-and-route functions.

Sometime around 1994, synthesis tools were equipped with knowledge about different FPGA architectures. This meant that they could perform mapping—and some level of packing—functions internally and output a LUT/CLB-level netlist. This netlist would subsequently be passed to the FPGA vendor's place-and-route software. The main advantage of this approach was that these synthesis tools had a better idea about timing estimations and area utilization, which allowed them to generate a better quality of results (QoR). In real terms, FPGA designs generated by architecturally aware synthesis tools were 15 to 20 percent faster than their counterparts created using traditional (gate-level) synthesis offerings.

Logic versus Physically Aware Synthesis

We're jumping a little bit ahead of ourselves here, but this is as good a place as any to briefly introduce this topic. The original logic synthesis tools were

designed for use with the multimicron ASIC technologies of the mid-1980s. In these devices, the delays associated with the logic gates far outweighed the delays associated with the tracks connecting those gates together. In addition to being relatively small in terms of gate-count (by today's standards), these designs featured relatively low clock frequencies and correspondingly loose design constraints. The combination of all of these factors meant that early logic synthesis tools could employ relatively simple algorithms to estimate the track delays, but that these estimations would be close enough to the real (post-place-and-route) values that the device would work.

Over the years, ASIC designs increased in size (number of gates) and complexity. At the same time, the dimensions of the structures on the silicon chip were shrinking with two important results:

- Delay effects became more complex in general.
- The delays associated with tracks began to outweigh the delays associated with gates.

By the mid-1990s, ASIC designs were orders of magnitude larger—and their delay effects were significantly more sophisticated—than those for which the original logic synthesis tools had been designed. The result was that the estimated delays used by the logic synthesis tool had little relation to the final post-place-and-route delays. In turn, this meant that achieving timing closure (tweaking the design to make it achieve its original performance goals) became increasingly difficult and time-consuming.

For this reason, ASIC flows started to see the use of *physically aware synthesis* somewhere around 1996. For the moment, we need only note that, during the course of performing its machinations, the physically aware synthesis engine makes initial placement decisions for the logic gates and functions. Based on these placements, the tool can generate more accurate timing estimations.

Ultimately, the physically aware synthesis tool outputs a placed (but not routed) gate-level netlist. The ASIC's physical implementation (place-and-route) tools use this initial placement information as a starting point from which to perform local (fine-grained) placement optimizations followed by detailed routing. The result is that the estimated delays used by the physically aware synthesis application more closely correspond to the post-place-and-route delays. In turn, this means that achieving timing closure becomes a less taxing process.

But what of FPGAs? Well, these devices were also increasing in size and complexity throughout the 1990s. By the end of the millennium, FPGA designers were running into significant problems with regard to timing closure. Thus, around 2000, EDA vendors started to provide FPGA-centric, physically aware synthesis offerings that could output a mapped, packed, and placed LUT/CLB-level netlist. In this case, the FPGA's physical implementation (place-and-route) tools use this initial placement information as a starting point from which to perform local (fine-grained) placement optimizations followed by detailed routing.

FAQ

Do FPGA designers still use graphical design entry?

When the first HDL-based flows appeared on the scene, many folks assumed that graphical design entry and visualization tools, such as schematic capture systems, were poised to exit the stage forever. Indeed, for some time, many design engineers prided themselves on using text editors like VI (from Visual Interface) or EMACS as their only design entry mechanism. But a picture tells a thousand words, as they say, and graphical entry techniques remain popular at a variety of levels. For example, it is extremely common to use a block-level schematic editor to capture the design as a collection of high-level blocks that are connected together. The system might then be used to automatically create a skeleton HDL framework with all of the block names and inputs and outputs declared. Alternatively, the user might create a skeleton framework in HDL, and the system might use this to create a block-level schematic automatically.

From the user's viewpoint, "pushing" down into one of these schematic blocks might automatically open an HDL editor. This could be a pure text-and-command–based editor like VI, or it might be a more sophisticated HDL-specific editor featuring the ability to show language keywords in different colors, automatically complete statements, and so forth.

Furthermore, when pushing down into a schematic block, modern design systems often give you a choice between entering and viewing the contents of that block as another, lower-level block-level schematic, raw HDL code, a graphical state diagram (used to represent an FSM), a graphical flow-chart, and so forth. In the case of the graphical representations like state diagrams and flowcharts, these can subsequently be used to generate their RTL equivalents automatically (Figure 5-17).

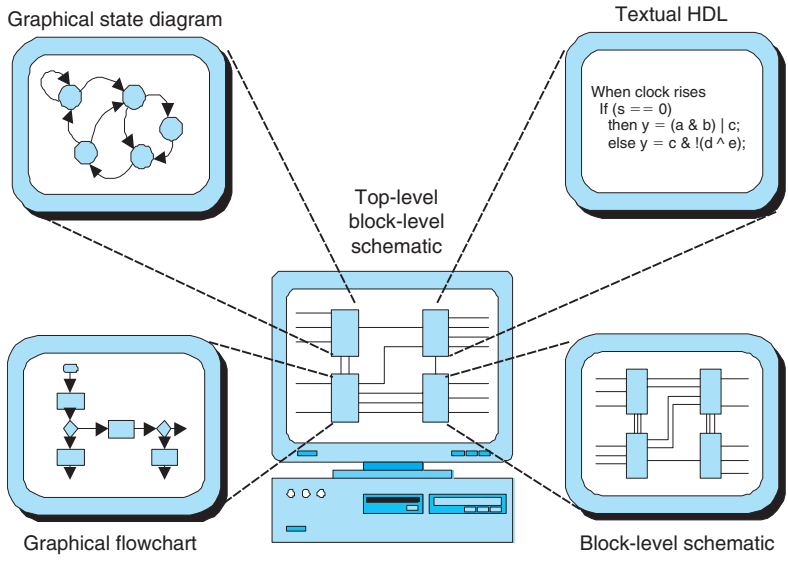


FIGURE 5-17 Mixed-level design capture environment.

Furthermore, it is common to have a tabular file containing information relating to the device's external inputs and outputs. In this case, both the top-level block diagram and the tabular file will (hopefully) be directly linked to the same data and will simply provide different views of that data. Making a change in any view will update the central data and be reflected immediately in all of the views.

A Plethora of HDLs

Life would be so simple if there were only a single HDL to worry about, but no one said that living was going to be easy. As previously noted, in the early days of digital IC electronics design (circa the 1970s), anyone who created an HDL-based design tool typically felt moved to create his or her own language to accompany it. Not surprisingly, the result was a morass of confusion (you had to be there to fully appreciate the dreadfulness of the situation). What was needed was an industry-standard HDL that could be used by multiple EDA tools and vendors, but where was such a gem to be found?

Verilog HDL

Sometime around the mid-1980s, Phil Moorby (one of the original members of the team that created the famous HILO logic simulator) designed a new HDL called Verilog. In 1985, the company he was working for, Gateway Design Automation, introduced this language to the market along with an accompanying logic simulator called Verilog-XL.

One very cool concept that accompanied Verilog and Verilog-XL was the Verilog programming language interface (PLI). The more generic name for this sort of thing is *application programming interface* (API). An API is a library of software functions that allow external software programs to pass data into an application and access data from that application. Thus, the Verilog PLI is an API that allows users to extend the functionality of the Verilog language and simulator.

As one simple example, let's assume that an engineer is designing a circuit that makes use of an existing module to perform a mathematical function such as a fast Fourier transform (FFT). A Verilog representation of this function might take a long time to simulate, which would be a pain if all the engineer really wanted to do was verify the new portion of the circuit. In this case, the engineer might create a model of this function in the C programming language, which would simulate, say, 1,000 times faster than its Verilog equivalent. This model would incorporate PLI constructs, allowing it to be linked into the simulation environment. The model could subsequently be accessed from the Verilog description of the rest of the circuit by means of a PLI call providing a bidirectional link to pass data back and forth between the main circuit (represented in Verilog) and the FFT (captured in C).

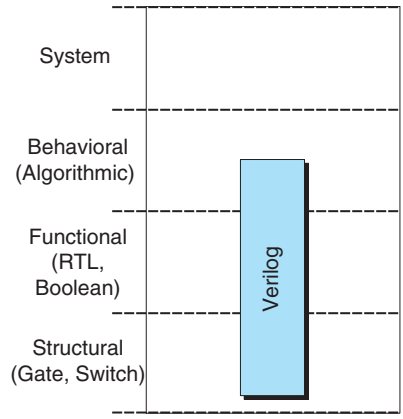


FIGURE 5-18 Levels of abstraction (Verilog).

Yet one more very useful feature associated with Verilog and Verilog-XL was the ability to have timing information specified in an external text file known as a *standard delay format* (SDF) file. This allowed tools like post-place-and-route timing analysis packages to generate SDF files that could be used by the simulator to provide more accurate results.

As a language, the original Verilog was reasonably strong at the structural (switch and gate) level of abstraction (especially with regard to delay modeling capability); it was very strong at the functional (Boolean equation and RTL) level of abstraction; and it supported some behavioral (algorithmic) constructs (Figure 5-18).

In 1989, Gateway Design Automation along with Verilog (the HDL) and Verilog-XL (the simulator) were acquired by Cadence Design Systems. The most likely scenario at that time was for Verilog to remain as just another proprietary HDL. However, with a move that took the industry by surprise, Cadence put the Verilog HDL, Verilog PLI, and Verilog SDF specifications into the public domain in 1990.

This was a very daring move because it meant that anybody could develop a Verilog simulator, thereby becoming a potential competitor to Cadence. The reason for Cadence's largesse was that the VHDL language (introduced later in this section) was starting to gain a significant following. The upside of placing Verilog in the public domain was that a wide variety of companies developing HDL-based tools, such as logic synthesis applications, now felt comfortable using Verilog as their language of choice.

—Technology Trade-offs—

- Having a single design representation that could be used by simulation, synthesis, and other tools made everyone's life much easier. It is important to remember, however, that Verilog was originally conceived with simulation

in mind; applications like synthesis were something of an afterthought. This means that when creating a Verilog representation to be used for both simulation and synthesis, one is restricted to using a synthesizable subset of the language (which is loosely defined as whatever collection of language constructs your particular logic synthesis package understands and supports).

The formal definition of Verilog is encapsulated in a document known as the language reference manual (LRM), which details the syntax and semantics of the language. In this context, the term *syntax* refers to the grammar of the language—such as the ordering of the words and symbols in relation to each other—while the term *semantics* refers to the underlying meaning of the words and symbols and the relationships between the things they denote.

In an ideal world, an LRM would define things so rigorously that there would be no chance of any misinterpretation. In the real world, however, there were some ambiguities with respect to the Verilog LRM. Admittedly, these were corner-case conditions along the lines of "if a control signal on this register goes inactive at the same time as the clock signal triggers, which signal will be evaluated by the simulator first?" But the result was that different Verilog simulators might generate different results, which is always somewhat disconcerting to the end user.

Verilog quickly became very popular. The problem was that different companies started to extend the language in different directions. In order to curtail this sort of thing, a nonprofit body called Open Verilog International (OVI) was established in 1991. With representatives from all of the major EDA vendors of the time, OVI's mandate was to manage and standardize Verilog HDL and the Verilog PLI.

The popularity of Verilog continued to rise exponentially, with the result that OVI eventually asked the IEEE to form a working committee to establish Verilog as an IEEE standard. Known as IEEE 1364, this committee was formed in 1993. May 1995 saw the first official IEEE Verilog release, which is formally known as IEEE 1364–1995, and whose unofficial designation has come to be Verilog 95.

Minor modifications were made to this standard in 2001; hence, it is often referred to as the Verilog 2001 (or Verilog 2K1) release. At the time of this writing, the IEEE 1364 committee is working feverishly on a forthcoming Verilog 2005 offering, while the design world holds its breath in dread anticipation (see also the section on "Superlog and System-Verilog" later in this chapter).

VHDL and VITAL

In 1980, the U.S. Department of Defense (DoD) launched the very high speed integrated circuit (VHSIC) program, whose primary objective was to advance the state of the art in digital IC technology. This program sought to address, among other things, the fact that it was difficult to reproduce ICs (and circuit boards) over the long life cycles of military equipment because the function

of the parts wasn't documented in a rigorous fashion. Furthermore, different components forming a system were often designed and verified using diverse and incompatible simulation languages and design tools.

To address these issues, a project to develop a new hardware description language called VHSIC HDL (or VHDL for short) was launched in 1981. One unique feature of this process was that industry was involved from a very early stage. In 1983, a team comprising Intermetrics, IBM, and Texas Instruments was awarded a contract to develop VHDL, the first official release of which occurred in 1985.

Also of interest is the fact that in order to encourage acceptance by the industry, the DoD subsequently donated all rights to the VHDL language definition to the IEEE in 1986. After making some modifications to address a few known problems, VHDL was released as official standard IEEE 1076 in 1987. The language was further extended in a 1993 release and again in 1999.

—Technology Trade-offs—

- As a language, VHDL is very strong at the functional (Boolean equation and RTL) and behavioral (algorithmic) levels of abstraction, and supports some system-level design constructs. However, VHDL is a little weak when it comes to the structural (switch and gate) level of abstraction, especially with regard to its delay modeling capability.

It quickly became apparent that VHDL had insufficient timing accuracy to be used as a sign-off simulator. For this reason, the VITAL initiative was launched at the Design Automation Conference (DAC) in 1992. VHDL Initiative toward ASIC Libraries (VITAL) was an effort to enhance VHDL's capabilities for modeling timing in ASIC and FPGA design environments. The result encompassed both a library of ASIC/FPGA primitive functions and an associated method for back-annotating delay information into these library models, where this delay mechanism was based on the same underlying tabular format used by Verilog (Figure 5-19).

Mixed-language Designs

Once upon a time, it was fairly common for an entire design to be captured using a single HDL (Verilog or VHDL). As designs increased in size and complexity, however, it became more common for different portions of the design to be created by different teams. These teams might be based in different companies or even reside in different countries, and it was not uncommon for the different groups to be using different design languages.

Another consideration was the increasing use of legacy design blocks or third-party IP, where the latter refers to a design team purchasing a predefined function from an external supplier. As a general rule of thumb related to Murphy's Law, if you were using one language, then the IP you wanted was probably available only in the other language.

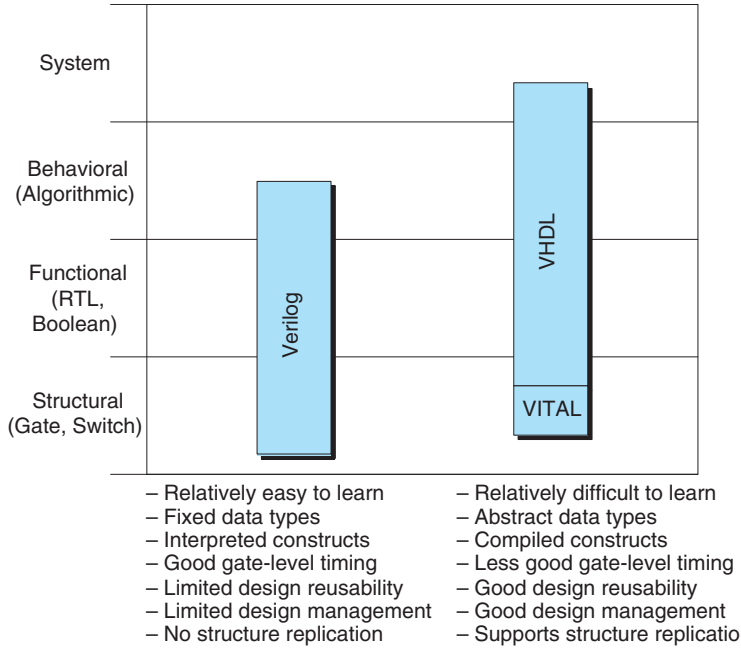


FIGURE 5-19 Levels of abstraction (Verilog versus VHDL).

The early 1990s saw a period known as the HDL Wars, in which the proponents of one language (Verilog or VHDL) stridently predicted the imminent demise of the other … but the years passed and both languages retained strong followings. The result was that EDA vendors began to support mixed-language design environments featuring logic simulators, logic synthesis applications, and other tools that could work with designs composed from a mixture of Verilog and VHDL blocks (or modules, depending on your language roots).

UDL/I

As previously noted, Verilog was originally designed with simulation in mind. Similarly, VHDL was created as a design documentation and specification language that took simulation into account. As a result, one can use both of these languages to describe constructs that can be simulated, but not synthesized. To address these problems, the Japan Electronic Industry Development Association (JEIDA) introduced its own HDL, the unified design language for integrated circuits (UDL/I) in 1990.

The key advantage of UDL/I was that it was designed from the ground up with both simulation and synthesis in mind. The UDL/I environment includes a simulator and a synthesis tool and is available for free (including the source code). However, by the time UDL/I arrived on the scene, Verilog and VHDL

already held the high ground, and UDL/I has never really managed to attract much interest outside of Japan.

Superlog and SystemVerilog

In 1997, things started to get complicated because that's when a company called Co-Design Automation was formed. Working away furiously, the folks at Co-Design developed a "Verilog on steroids" called Superlog.

Superlog was an amazing beast that combined the simplicity of Verilog with the power of the C programming language. It also included things like temporal logic, sophisticated design verification capabilities, a dynamic API, and the concept of assertions that are key to the formal verification strategy known as model checking. (VHDL already had a simple assert construct, but the original Verilog had nothing to boast about in this area.)

The two main problems with Superlog were

- it was essentially another proprietary language, and
- it was so much more sophisticated than Verilog 95 (and later Verilog 2001) that getting other EDA vendors to enhance their tools to support it would have been a major feat.

Meanwhile, while everyone was scratching their heads wondering what the future held, the OVI group linked up with their equivalent VHDL organization called VHDL International to form a new body called Accellera. The mission of this new organization was to focus on identifying new standards and formats, to develop these standards and formats, and to foster the adoption of new methodologies based on these standards and formats.

In the summer of 2002, Accellera released the specification for a hybrid language called SystemVerilog 3.0 (don't even ask me about 1.0 and 2.0). The great advantage to this language was that it was an incremental enhancement to the existing Verilog, rather than the death-defying leap represented by a full-up Superlog implementation. Actually, SystemVerilog 3.0 featured many of Superlog's language constructs donated by Co-Design. It included things like the assertion and extended synthesis capabilities that everyone wanted and, being an Accellera standard, it was well placed to quickly gain widespread adoption.

The current state of play (at the time of this writing) is that Co-Design was acquired by Synopsys in the fall of 2002. Synopsys maintained the policy of donating language constructs from Superlog to SystemVerilog, but no one is really talking about Superlog as an independent language anymore. After a little pushing and pulling, all of the mainstream EDA vendors officially endorsed SystemVerilog and augmented their tools to accept various subsets of the language, depending on their particular application areas and requirements. System-Verilog 3.1 hit the streets in the summer of 2003, followed by a 3.1a release (to add a few enhancements and fix some annoying problems) around

the beginning of 2004. Meanwhile, the IEEE determined to release the next version of Verilog in 2005. To avert a potential schism between Verilog 2005 and SystemVerilog, Accellera promised to donate their SystemVerilog copyright to the IEEE by the summer of 2004. SystemVerilog was formally adopted as IEEE Standard 1800–2005. At the time of writing, the IEEE is working on the next major version of the standard, expected as 1800–2008. They are also extending the APIs to include assertions, coverage, and other aspects of the language.

Speaking of which … there is another aspect to SystemVerilog, the full potential of which has not yet been realized. This is the Direct Programming Interface (DPI). In fact, the concept behind this is incredibly simple. Since processes in Verilog look very much like procedure calls in C, why not make them able to call each other directly without having to go through a massive interface as was the case with the Verilog PLI? The resulting interface is extremely fast (although hidden dangers can lie there) and means that SystemVerilog now plays nicely with other languages, such as SystemC. In effect, that means that the SystemVerilog language has become more extensible.

SystemC

And then we have SystemC, which some design engineers love and others hate with a passion. SystemC—discussed in more detail in Chapter 6—can be used to describe designs at the RTL level of abstraction. These descriptions can subsequently be simulated 5 to 10 times faster than their Verilog or VHDL counterparts, and synthesis tools are available to convert the SystemC RTL into gate-level netlists.

—Technology Trade-offs—

- One big argument for SystemC is that it provides a more natural environment for hardware/software codesign and co-verification.
- One big argument against it is that the majority of design engineers are very familiar with Verilog or VHDL, but are not familiar with the object-oriented aspects of SystemC.
- Another consideration is that the majority of today's synthesis offerings represent hundreds of engineer years of development in translating Verilog or VHDL into gate-level netlists. By comparison, there are far fewer SystemC-based synthesis tools, and those that are available tend to be somewhat less sophisticated than their more traditional counterparts.

In reality, SystemC is more applicable to a system-level versus an RTL design environment. Having said this, SystemC seems to be gaining a lot of momentum in Asia and Europe, and the debate on SystemC versus SystemVerilog versus VHDL will doubtless be with us for quite some time.

Points to Ponder

Sad to relate, the majority of designs described in RTL are almost unintelligible to another designer. In an ideal world, the RTL description of a design should read like a book, starting with a "table of contents" (an explanation of the design's structure), having a logical flow partitioned into "chapters" (logical breaks in the design), and having lots of "commentary" (comments explaining the structure and operation of the design).

It's also important to note that coding style can impact performance (this typically affects FPGAs more than ASICs). One reason for this is that, although they might be logically equivalent, different RTL statements can yield different results. Also, tools are part of the equation because different tools can yield different results.

The various FPGA vendors and EDA vendors are in a position to provide their customers with reams of information on particular coding styles and considerations with regard to their chips and tools, respectively. However, the following points are reasonably generic and will apply to most situations.

Serial versus Parallel Multiplexers

When creating RTL code, it is useful to understand what your synthesis tool is going to do in certain circumstances. For example, every time you use an if-then-else statement, the result will be a 2:1 multiplexer. This becomes interesting in the case of nested if-then-else statements, which will be synthesized into a priority structure. For example, assume that we have already declared signals Y, A, B, C, D, and SEL (for select) and that we use them to create a nested if-then-else (Figure 5-20).

```
if     SEL == 00" then Y = A;
elseif SEL == 01" then Y = B;
elseif SEL == 10" then Y = C;
else                   Y = D;
end if;
```

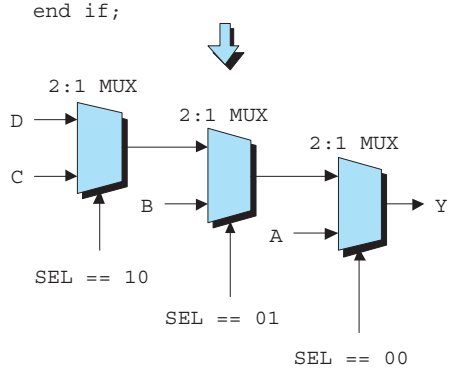


FIGURE 5-20 Synthesizing nested if-then-else statements.

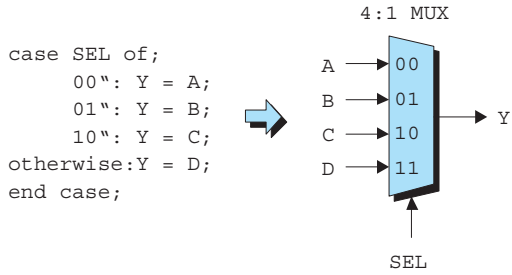


FIGURE 5-21 Synthesizing a case statement.

As before, the syntax used here is a generic one that doesn't really reflect any of the mainstream languages. In this case, the innermost if-then-else will be the fastest path, while the outermost if-then-else will be the critical signal (in terms of timing). Having said this, in some FPGAs all of the paths through this structure will be faster than using a case statement. Speaking of which, a case statement implementation of the above will result in a 4:1 multiplexer, in which all of the timing paths associated with the inputs will be (relatively) equal (Figure 5-21).

Beware of Latch Inference

Generally speaking, it's a good idea to avoid the use of latches in FPGA designs unless you really need them. One other thing to watch out for: If you use an if-then-else statement, but neglect to complete the "else" portion, then most synthesis tools will infer a latch.

Use Constants Wisely

Adders are the most used of the more complex operators in a typical design. In certain cases, ASIC designers sometimes employ special versions using combinations of half-adders and full-adders. This may work very efficiently in the case of a gate array device, for example, but it will typically result in a very bad FPGA implementation.

When using an adder with constants, a little thought goes a long way. For example, "A + 2" can be implemented more efficiently as "A + 1 with carry-in," while "A − 2" would be better implemented as "A − 1 with carry-in."

Similarly, when using multipliers, "A * 2" can be implemented much more efficiently as "A SHL 1" (which translates to "A shifted left by one bit"), while "A * 3" would be better implemented as "(A SHL 1) + A."

In fact, a little algebra also goes a long way in FPGAs. For example, replacing "A * 9" with "(A SHL 3) + A" results in at least a 40-percent reduction in area.

Consider Resource Sharing

Resource sharing is an optimization technique that uses a single functional block (such as an adder or comparator) to implement several operators in the HDL

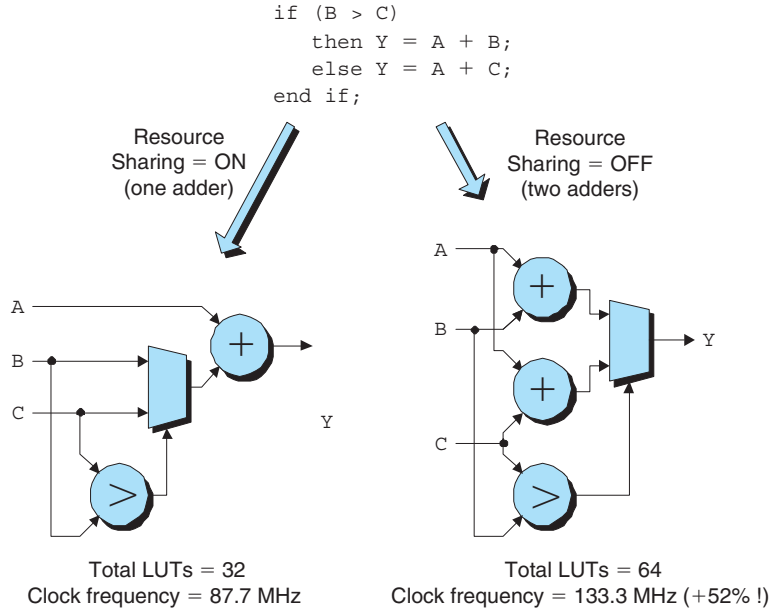


FIGURE 5-22 Resource sharing.

code. If you do not use resource sharing, then each RTL operation is built using its own logic. This results in better performance, but it uses more logic gates, which equates to silicon real estate. If you do decide to use resource sharing, the result will be to reduce the gate-count, but you will typically take a hit in performance. For example, consider the statement illustrated in Figure 5-22. Note that frequency values shown in this figure are of interest only for the purposes of this comparison, because these values will vary according to the particular FPGA architecture, and they will change as new process nodes come online.

The following operators can be shared with other instances of the same operator or with related operators on the same line:

For example, a + operator can be shared with instances of other + operators or with – operators, while a * operator can be shared only with other * operators.

If nothing else, it's a good idea to check whether your synthesis application has resource sharing enabled or disabled by default. And one final point is that resource sharing in ASICs can alleviate routing congestion, but it may actually cause routing problems in FPGAs.

Last But Not Least

Internal tri-state buses are slow in most FPGAs and should be avoided unless you are 100-percent confident that you know what you're doing. If at all possible, use tri-state buffers only at the top-most level of the design. If you do wish

to use internal tri-state buffers, then in the case of FPGA families that don't support these gates, the majority of today's synthesis tools provide automatic tri-state-to-multiplexer conversion (this basically involves converting the tri-state buffers specified in the RTL into corresponding LUT/CLB-based logic).

Also, bidirectional buffers can cause timing loop problems, so if you use them, make sure that any false paths are clearly marked.

INSTANT SUMMARY

Table 5-1 summarizes the features of the main HDLs as related to FPGA design.

TABLE 5.1 Summary of Major HDL Features

Verilog 2005	Syntax similar to C; API/PLI support; Timing specified in external text (SDF) file; Strong at structural level of abstraction especially w/ delay modeling; Strong at functional level of abstraction; Supports some behavioral constructs
SystemVerilog	Superset of Verilog 2005 w/ many new features to aid design; Verification and design modeling; Assertion and extended synthesis capability
VHDL/VITAL	Strong at functional and behavioral levels of abstraction; Supports some system-level design constructs Somewhat weak on structural level of abstraction esp. regarding delay modeling; VITAL enhances abilities for modeling timing in ASIC and FPGA design environments
SystemC	More of a system description language; Implemented in C++; Can describe design at RTL level of abstraction, and these designs can be simulated 5–10 times faster than Verilog or VHDL counterparts; Synthesis tools are available to convert SystemC RTL into gate-level netlists More natural for hardware/software co-design

Chapter 6

Other Design Flows

In an Instant

C/C++–based Design Flows
- C versus C++ and Concurrent versus Sequential
- SystemC-based Flows
- Augmented C/C++-based Flows
- Pure C/C++-based Flows
- Different Levels of Synthesis Abstraction
- Mixed-language Design and Verification Environments

DSP-based Design Flows
- Alternative DSP Implementations

FPGA-centric Design Flows for DSPs
- Mixed DSP and VHDL/Verilog etc. Environments

Embedded Processor-based Design Flows
- Hard versus Soft Cores
- Partitioning a Design into Its Hardware and Software Components
- Using an FPGA as Its Own Development Environment
- Improving Visibility in the Design
- A Few Coverification Alternatives

Instant Summary

Definitions

Again we'll start with some basic terms and their definitions.

- *Microarchitecture definition* tasks include such things as detailing control structures, bus structures, and primary data path elements.
- You were introduced to SystemC in the last chapter, but here we'll go into more detail on this C++-based *system description language*.
- *Pragmas* are commented directives or special comments that can be put into pure C code to extend its capabilities, such as for use in FPGA design flows.
- We'll look at *digital signal processing* (DSP) based design flows in this chapter, which refers to the branch of electronics concerned with the representation and manipulation of signals in digital form.

- *Domain-specific languages* (DSLs) are languages, such as MATLAB, that provide more concise ways of representing specific tasks than do general-purpose languages.
- A *microcontroller* combines a CPU core with selected peripherals and specialized inputs and outputs.
- A *hard microprocessor core* is a core that is implemented as a dedicated, predefined (hardwired) block.
- A *soft core* is a group of programmable logic blocks configured to act as a microprocessor.
- An *instruction set simulator* (ISS) provides a virtual representation of a CPU being implemented.
- A *bus interface model* (BIM) is an entity that acts as a translator between the simulator and the ISS.

C/C++-BASED DESIGN FLOWS

With regard to the traditional HDL-based flows introduced in Chapter 5, a design commences with an original concept, whose high-level definition is determined by system architects and system designers. It is at this stage that macro-architecture decisions are made, such as partitioning the design into hardware and software components.

The resulting specification is then handed over to the hardware design engineers, who commence their portion of the development process by performing *microarchitecture definition tasks* such as detailing control structures, bus structures, and primary data path elements. These microarchitecture definitions, which are often performed in brainstorming sessions on a whiteboard, may include performing certain operations in parallel versus sequential, pipelining portions of the design versus nonpipelining, sharing common resources (for example, two operations sharing a single multiplier, versus using dedicated resources) and so forth.

Eventually, the design intent is captured by writing RTL VHDL/Verilog. Following verification via simulation, this RTL is then synthesized down to a structural netlist suitable for use by the target technology's place-and-route applications (Figure 6-1).

At the time of this writing, these VHDL or Verilog-based flows account for around 95 percent of all ASIC and FPGA designs; however, there are a number of problems associated with these flows:

- *Capturing the RTL is time-consuming*: Even though Verilog and VHDL are intended to represent hardware, it is still time-consuming to use these languages to capture the functionality of a design.
- *Verifying RTL is time-consuming*: Using simulation to verify large designs represented in RTL is computationally expensive and time-consuming.

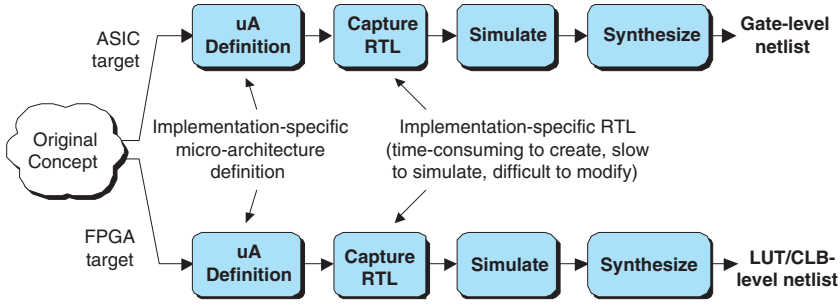


FIGURE 6-1 Traditional (simplified) HDL-based flows.

- *Evaluating alternative implementations is difficult*: Modifying and reverifying RTL to perform a series of what-if evaluations of alternative microarchitecture implementations is difficult and time-consuming. This means that the number of evaluations the design team can perform may be limited, which can result in a less-than-optimal implementation.
- *Accommodating specification changes is difficult*: If any changes to the specification are made during the course of the project, folding these changes into the RTL and performing any necessary reverification can be painful and time-consuming. This is a significant consideration in certain application areas, such as wireless projects, because broadcast standards and protocols are constantly evolving and changing.
- *The RTL is implementation specific*: Realizing a design in an FPGA typically requires a different RTL coding style from that used for an ASIC implementation. This means that it can be extremely difficult to retarget a complex design represented in RTL from one implementation technology to another. This is of concern when one is migrating an existing ASIC design into an FPGA equivalent or creating an FPGA design to be used as a prototype for a future ASIC implementation.

One way to view this is that all of the implementation intelligence associated with the design is hardcoded into the RTL, which therefore becomes implementation specific. It's important to understand that this implementation specificity goes beyond the coarse ASIC-versus-FPGA boundary, which dictates that RTL intended for an FPGA implementation is not suitable for an optimal ASIC realization, and vice versa. Even assuming a single target device architecture, the way in which a set of algorithms is used to process data may require a number of different microarchitecture implementations, depending on the target application areas.

Actually, to be scrupulously fair, we should probably note that the same RTL may be used to drive both ASIC and FPGA implementations. The reason for doing this is to avoid the risk of introducing a functional bug into the RTL when retargeting the code, but there is typically a penalty to be paid. That is,

if code originally targeted toward an FPGA implementation is subsequently used to drive an ASIC implementation, the resulting ASIC will typically require more silicon real estate and have higher power consumption as compared to using RTL created with an ASIC architecture in mind. Similarly, if code originally targeted toward an ASIC implementation is subsequently used to drive an FPGA implementation, the ensuing FPGA will typically take a significant performance hit as compared to using RTL created with an FPGA architecture in mind.

RTL is less than ideal for hardware-software codesign: System-on-chip (SoC) devices are generally understood to be those that include microprocessor cores. Irrespective of whether these designs are to be realized using ASICs or FPGAs, today's SoCs are exhibiting an ever-increasing amount of software content. When coupled with increased design reuse on the hardware side, in many cases it is necessary to verify the software and hardware concurrently so as to completely validate such things as the system diagnostics, RTOS, device drivers, and embedded application software. Generally speaking, it can be painful verifying (simulating) the hardware represented in VHDL or Verilog in conjunction with the software represented in C/C++ or assembly language.

One approach that addresses the issues enumerated above is to perform the initial design capture at a higher level of abstraction than can be achieved with RTL VHDL/Verilog. The first such level is to use some form of C/C++, but as usual nothing is simple because there are a variety of alternatives, including SystemC, augmented C/C++, and pure C/C++.

C versus C++ and Concurrent versus Sequential

Before we leap into the fray, we should tie down a couple of points to ensure that we're all marching in step to the same beat. First, there is a wide variety of programming languages available, but—excepting specialist application areas—the most commonly used by far are traditional C and its object-oriented offspring C++. For our purposes here, we will refer to these collectively as C/C++.

The next point of import is that, by default, statements in languages like C/C++ are executed sequentially. For example, assuming that we have already declared three integer variables called a, b, and c, then the following statements:

```
a = 6; /* Statement in C/C++ program */
b = 2; /* Statement in C/C++ program */
c = 9; /* Statement in C/C++ program */
```

would, perhaps not surprisingly, occur one after the other. However, this has certain implications; for example, if we now assume that the following statements occur sometime later in the program:

```
a = b; /* Statement in C/C++ program */
b = a; /* Statement in C/C++ program */
```

then *a* (which initially contained 6) will be loaded with the value currently stored in *b* (which is 2). Next, *b* (which initially contained 2) will be loaded with the value currently stored in *a* (which is now 2), so both *a* and *b* will end up containing the same value.

The sequential nature of programming languages is the way in which software engineers think. However, hardware design engineers have quite a different view of the world. Let's assume that a piece of hardware contains two registers called *a* and *b* that are driven by a common clock signal. Let's further assume that these registers have previously been loaded with values of 6 and 2, respectively. Finally, let's assume that at some point in the HDL code, we see the following statements:

```
a = b; /* Statement in VHDL/Verilog Code */
b = a; /* Statement in VHDL/Verilog Code */
```

As usual, this syntax doesn't actually represent VHDL or Verilog; it's just a generic syntax used only for the purposes of this example. Generally speaking, hardware engineers would expect both of these statements to be executed concurrently (at the same time). This means that *a* (which initially contained 6) will be loaded with the value stored in *b* (which was 2) while—at the same time—*b* (which initially contained 2) will be loaded with the value stored in *a* (which was 6). The result is that the initial contents of *a* and *b* will be exchanged.

As usual, of course, the above is something of a simplification. However, it's fair to say that HDL statements will execute concurrently by default, unless sequential behavior is forced by means of techniques like blocking assignments. Thus, by default, RTL-based logic simulators will execute the statements shown above in this concurrent manner; similarly RTL-based logic synthesis tools will generate hardware that handles these two activities simultaneously. By comparison, unless explicitly directed to do otherwise (by means of the techniques introduced later in this chapter), C/C++ statements will execute sequentially.

SystemC-based Flows

FAQs

What exactly is SystemC (and where did it come from)?

Before we consider SystemC-based flows, it is probably a good idea to elaborate a bit more on just what SystemC is, because there is typically some confusion on this point .

SystemC 1.0 – One of the underlying concepts behind SystemC is that it is an open-source environment to which everyone contributes. As an example, consider Linux, which was rough around the edges at first. Based on contributions from different folks, however, Linux eventually became a real operating system (OS) with the potential to challenge Microsoft. In this spirit, a relatively undocumented SystemC 1.0 was let loose to roam wild and free circa 2000. SystemC 1.0 was a C++ class library that facilitated the representation of notions such as concurrency (things happening at the same time), timing, and I/O pins. By means of this class library, engineers could capture designs at the RTL level of abstraction.

One advantage of this early incarnation was that it facilitated hardware/software codesign environments. Another was that SystemC representations at the RTL level of abstraction might simulate 5 to 10 times faster than their VHDL and Verilog counterparts. On the downside, it was harder and more time-consuming to capture an RTL-level design in SystemC 1.0 than with VHDL or Verilog. Furthermore, there was a scarcity of design tools that could synthesize SystemC 1.0 representations into netlist-level equivalents with any degree of sophistication.

SystemC 2.0 – Later, in 2002, SystemC 2.0 arrived on the scene. This augmented the 1.0 release with some high-level modeling constructs such as FIFOs (a form of memory that can accept and subsequently make available a series of words of data and that operates on a first-in first-out principle). The 2.0 release also included a variety of behavioral, algorithmic, and system-level modeling capabilities, such as the concepts of transactions and channels (which are used to describe the communication of data between blocks at an abstract level).

To gain a little more perspective on SystemC, let's first consider a typical scenario of how things would have worked using the original SystemC 1.0. As a simple example, let's assume that we have two functions called $f(x)$ and $g(x)$ that have to communicate with each other (Figure 6-2).

In this case, the interface between the blocks would have to be defined at the pin level. The real problem with this approach occurs when you are in the early stages of a design, because you are already defining implementation details such as bus widths. This makes things difficult to change if you wish to experiment with different what-if architectural scenarios. This aspect of things became much easier with SystemC 2.0, which allowed abstract interfaces to be declared between the blocks (Figure 6-3).

Now, the interfacing between the blocks can be performed at the level of abstract records on the basis that, in the early stages of the design cycle, we

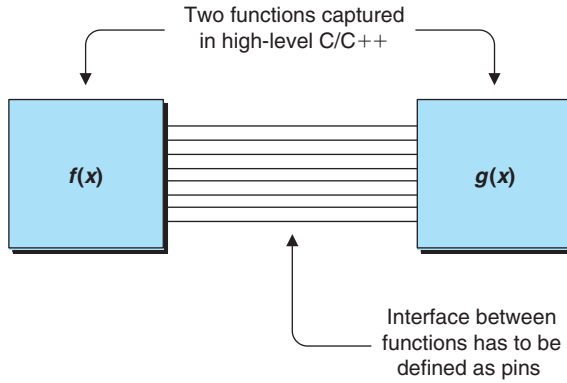


FIGURE 6-2 Interfacing in SystemC 1.0.

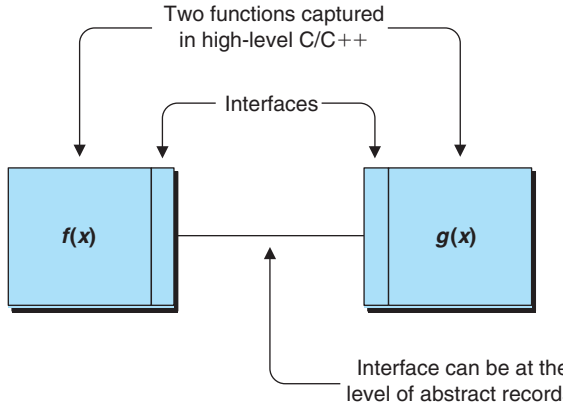


FIGURE 6-3 Interfacing in SystemC 2.0.

don't really care how data gets from point a to point b, just that it does get there somehow.

These abstract interfaces facilitate performing architectural evaluation early in the design cycle. Once the architecture starts to firm up, you can start refining the interface by using high-level constructs such as a FIFO to which one would assign attributes like width and depth and characteristics like blocking write, nonblocking read, and how to behave when empty or full. Still later, this logical interface can be replaced by a completely specified (pin-level) interface that binds the functional blocks together at a more physical level.

Levels of Abstraction

Truth to tell, this is where things start to become a little fuzzy around the edges, not the least because one runs into different definitions depending on to

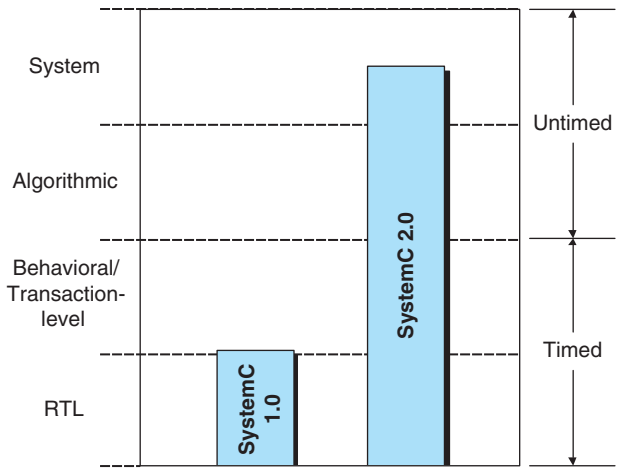


FIGURE 6-4 Levels of SystemC abstraction.

whom one is talking. As a first pass, however, we might take a stab at capturing the different levels of SystemC abstraction, as shown in Figure 6-4.

This is why things become confusing, because SystemC can mean all things to all people. To some it's a replacement for RTL VHDL/Verilog, while to others it's a single language that can be used for system-level specification, algorithmic and architectural analysis, behavioral design, and testbenches for use in verification.

One area of confusion comes when you start to talk about behavioral synthesis. This encompasses certain aspects of both the algorithmic and transactional levels (in the latter case, however, you have to be careful as to how to define your transactions).

SystemC-based Design-flow Alternatives

This is a tricky one because one might go various ways here.

- Many of today's designs begin life as complex algorithms. In this case, it is very common to start by creating a C or C++ representation. This representation can be used to validate the algorithms by compiling it into a form that can be run (simulated) 1,000 or more times faster than an RTL equivalent. In the case of the HDL-based flows discussed in Chapter 5, this C/C++ representation of the algorithms would then be hand-translated into RTL VHDL/Verilog. The C/C++ representation will typically continue to be used as a golden model, which means it can be linked into the RTL simulator and run in parallel with the RTL simulation. The results from the C/C++ and RTL models can be compared so as to ensure that they are functionally equivalent.

- Alternatively, in one flavor of a SystemC-based flow, the original C/C++ model could be incrementally modified by adding timing, concurrency, pin definitions, and so forth to transform it to a level at which it would be amenable to SystemC-based RTL or behavioral synthesis.
- In another flavor of a SystemC-based flow, the design might be initially captured in SystemC using system, algorithmic, or transaction-level constructs that could be used for verification at a high level of abstraction. This representation could then be incrementally modified to bring it down to a level at which it would be amenable to SystemC-based RTL or behavioral synthesis.

Irrespective of the actual route by which one might get there, let's assume that we are in possession of a SystemC representation of a design that is suitable for SystemC-based behavioral or RTL synthesis. In this case, there are two main design-flow alternatives, which are:

1. to **translate the System C into RTL VHDL/Verilog automatically** and then to use conventional RTL synthesis technology, or
2. to use **SystemC-based synthesis to generate an implementation-level netlist** directly.

—Technology Trade-offs—

- There are two schools of thought here. One says that synthesizing the SystemC directly into the implementation-level netlist offers the cleanest, fastest, and most efficient route.
- Another view is that it's better to translate the SystemC into RTL VHDL/Verilog first because RTL is the way design engineers really visualize their world; that this level is a natural staging point for integrating design blocks (including third-party IP) originating from multiple sources; and that Verilog/VHDL synthesis technology is extremely mature and powerful (as compared to SystemC-based synthesis technology).

Both of these flows can be applied to ASIC or FPGA targets (Figure 6-5).

The first SystemC synthesis applications were predominantly geared toward ASIC flows, so they didn't do a very good job at inferring FPGA-specific entities such as embedded RAMs, embedded multipliers, and so forth. More recent incarnations do a much better job of this, but the level of sophistication exhibited by different tools is a moving target, so the prospective user is strongly advised to perform some indepth evaluations before slapping a bundle of cash onto the bargaining table.

Note that Figure 6-5 shows the use of implementation specific SystemC to drive the ASIC versus FPGA flows. As soon as you start coding at the RTL level and adding timing concepts, be it in VHDL, Verilog, or SystemC, then achieving an optimal implementation requires that the code be written with a specific target architecture in mind.

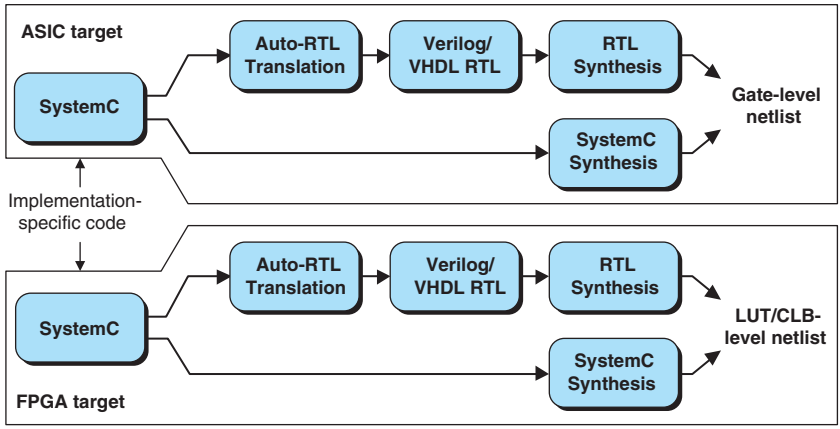


FIGURE 6-5 Alternative SystemC flows.

—*Technology Trade-offs*—

- Once again, having said this, the same SystemC can be used to drive both ASIC and FPGA flows, but there is typically a penalty to be paid. If SystemC code originally targeted toward an FPGA implementation is subsequently used to drive an ASIC flow, the resulting ASIC will typically require more silicon real estate and have higher power consumption as compared to using code created with an ASIC architecture in mind.
- Similarly, if code originally targeted toward an ASIC implementation is subsequently used to drive an FPGA flow, the ensuing FPGA will typically take a significant performance hit as compared to using code created with an FPGA architecture in mind. This is primarily a result of hard-coding the microarchitecture definition in the source.

Insider Info

Depending on who you are talking to, folks either love SystemC or they loath it. Most would agree that SystemC 2.0 is very promising and that there's no other language that provides the same capabilities (some of these capabilities are being added into SystemVerilog, but not all of them).

On the downside, many design engineers are reasonably proficient at writing C, but most of them are significantly less familiar with the object-oriented aspects of C++. So requiring them to use SystemC means giving them more power on the one hand, while thrusting them into a world they don't like or understand on the other. It's also true that while SystemC can be very useful for verification and high-level system modeling, in some respects it's still relatively immature toolwise with regard to actual implementation flows.

One school of thought says that, although SystemC is difficult to write by hand and also difficult to synthesize, which makes it a somewhat clumsy specification language, it does provide a powerful framework for simulation across languages and levels of abstraction. At the time of this writing, a number of companies that were strong supporters of SystemC in the United States have grown somewhat less vocal over the last few years. On the other hand, SystemC is gaining some ground in Europe and Asia. What does the future hold? Wait a few years, and I'll be happy to tell you!

Augmented C/C++-based Flows

FAQs

What do we mean by augmented C/C++?

There are two ways in which standard C/C++ can be augmented to extend its capabilities and the things it can be used to represent. The first is to include special comments, known as commented directives or pragmas, into the pure C/C++ code. These comments can subsequently be recognized and interpreted by parsers, precompilers, compilers, and other tools and used to add constructs to the code or modify the way in which it is processed. One significant drawback to this approach is that simulation requires the use of proprietary C/C++ compilers as opposed to using standard off-the-shelf compilers. This limits the options customers have and is only viable if standards are developed for multiple EDA vendors to leverage.

The other way in which C/C++ can be augmented is to add special keywords and statements into the language. This is a very popular technique, and there is a veritable plethora of such language variants roaming wild and free around the world, each tailored toward a different application area. One downside of this approach is that, once again, it requires proprietary C/C++ compilers; otherwise, tools such as simulators that have not been enhanced to understand these new keywords and statements will crash and burn. A common solution to this problem is to wrap standard #ifdef directives around the new keywords and statements such that a precompiler can be used to discard them as required (this is somewhat inelegant, but it works).

In the case of capturing the functionality of hardware for ASIC and FPGA designs, it is necessary to augment standard C/C++ with special statements to support such concepts as clocks, pins, concurrency, synchronization, and resource sharing.

Assuming that you have an initial model represented in pure C/C++, the first step would be to augment it with clock statements, along with interface statements used to define the input and output pins. You could then use an appropriate synthesis tool to generate an implementation (as discussed below).

However, because C/C++ is by nature sequential, the resulting hardware can be horribly slow and inefficient if the synthesis tool is not capable of locating potential parallelisms and exploiting them.

For example, assume that we have the following statements in a C/C++ representation of the design:

```
a = 6;       /* Standard C/C++ statement */
b = 2;       /* Standard C/C++ statement */
c = 9;       /* Standard C/C++ statement */
d = a + b;   /* Standard C/C++ statement */
:
etc
```

By default, each = sign is assumed by the synthesis application to represent one clock cycle. Thus, if the above code were left as is, the augmented C/C++ synthesis tool would generate hardware that loaded variable (register) *a* with 6 on the first clock, then *b* with 2 on the next clock, then *c* with 9 on the next clock, and so forth. Thus, by hardware standards, this would run horribly slowly.

Of course, most synthesis tools would be capable of locating and exploiting the potential parallelisms in the above example, but they might well miss more complex cases that require human consideration and intervention. For the purposes of these discussions, however, we shall continue to work with this simple test case. The point is that an augmented C/C++ language will have keywords like "parallel" (or "par") and "sequential" (or "seq") that will instruct the downstream synthesis application as to which statements should be executed in parallel, and so forth. For example:

```
parallel;      /* Augmented C/C++ statement */
a = 6;         /* Standard C/C++ statement */
b = 2;         /* Standard C/C++ statement */
c = 9;         /* Standard C/C++ statement */
sequential;    /* Augmented C/C++ statement */
d = a + b;     /* Standard C/C++ statement */
:
etc
```

In this case, the *parallel* statement instructs the synthesis tool that the following statements can be implemented concurrently, while the *sequential* statement implies that the preceding operations must occur prior to any subsequent actions taking place. Of course, these parallel and sequential statements can be nested as required.

Things become more complex in the case of loops, depending on whether the designer wishes to unravel them partially or fully. Just to give a point of reference, we might visualize a loop as being something like "*for i = 1 to 10 in increments of 1 do xxxx, yyyy, and zzzz*". In some cases, it may be possible to simply associate a parallel or sequential statement with the loop, but if more subtlety is required, the designer may be obliged to completely rewrite these constructs.

It may also be necessary to add "share" statements if resource sharing is required, and "channel" statements to share signals between expressions, and the list goes on.

ALERT!

As was previously noted, tools such as simulators that have not been enhanced to understand these new keywords and statements will "crash-and-burn" when presented with this representation. One solution is to "wrap" standard "#ifdef" directives around the new keywords and statements such that a precompiler can be used to discard them as required. However, this means that the simulator and synthesis engines will be working on different views of the design, which is typically not a good idea. The other solution is to use a proprietary simulator, but this may not have the power, capacity, or capabilities of your existing simulation technology.

Augmented C/C++ Design-flow Alternatives

As usual, one might go various ways here. As we previously discussed, in the case of a design that begins life as a suite of algorithms, it is very common to start by creating a C or C++ representation. Following verification, this C/C++ model can be incrementally modified by adding statements for clocks, pins, concurrency, synchronization, and resource sharing so as to make the model suitable for the appropriate synthesis utility. Alternatively, the design might be captured using the augmented C/C++ language from the get-go.

Irrespective of the actual route we might take to get there, let's assume that we are in possession of an augmented C/C++ representation of a design that is suitable for synthesis. Once again, there are two main design-flow alternatives, which are (1) to translate the augmented C/C++ into Verilog or VHDL at the RTL level of abstraction automatically and to then use conventional RTL synthesis technology, or (2) to use an appropriate augmented C/C++ synthesis engine.

And, once again, one school of thought says that synthesizing the augmented C/C++ directly into the implementation level netlist offers the cleanest, fastest, and most efficient route. Others say that the RTL Verilog/VHDL level is the natural staging post for design integration and that today's RTL synthesis technology is extremely mature and powerful.

Both of these flows can be applied to ASIC or FPGA targets (Figure 6-6). The first augmented C/C++ synthesis applications were predominantly geared

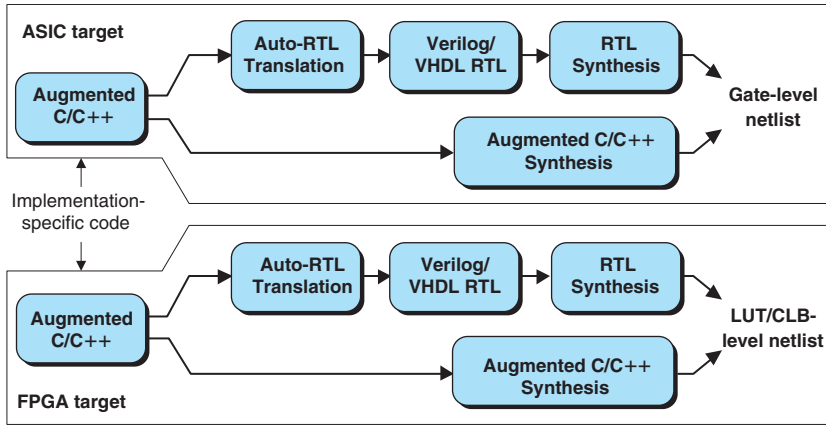


FIGURE 6-6 Alternative augmented C/C++ flows.

toward ASIC flows. This meant that these early incarnations didn't do a tremendous job when it came to inferring FPGA-specific entities such as embedded RAMs, embedded multipliers, and so forth. More recent versions of these tools do a much better job at this, but, as usual, the prospective user is strongly advised to perform some in-depth evaluations before handing over any hard-earned cash.

Note that Figure 6-6 shows the use of implementation-specific code to drive the ASIC versus FPGA flows because achieving an optimal implementation requires that the code be written with a specific target architecture in mind. In reality, the same code can be used to drive both ASIC and FPGA flows, but there is usually a penalty to be paid (see the discussions on SystemC for more details).

Pure C/C++-based Flows

Last, but not least, we come to pure C/C++-based flows. In reality, the term *pure C/C++* actually refers to industry-standard C/C++ that is minimally augmented with SystemC data types to allow specific bit widths to be associated with variables and constants.

Although relatively new, pure C/C++-based flows offer a number of advantages as compared to other C-based flows and traditional Verilog-/VHDL-based flows:

- *Creating pure C/C++ is fast and efficient*: Pure untimed C/C++ representations are more compact and easier to create and understand than equivalent SystemC and augmented C/C++ representations (and they are much more compact than their RTL equivalents, requiring perhaps 1/10th to 1/100th of the code).
- *Verifying C/C++ is fast and efficient*: A pure untimed C/C++ representation will simulate significantly faster than a timed SystemC or augmented C/C++ model and 100 to 10,000 times faster than an equivalent RTL

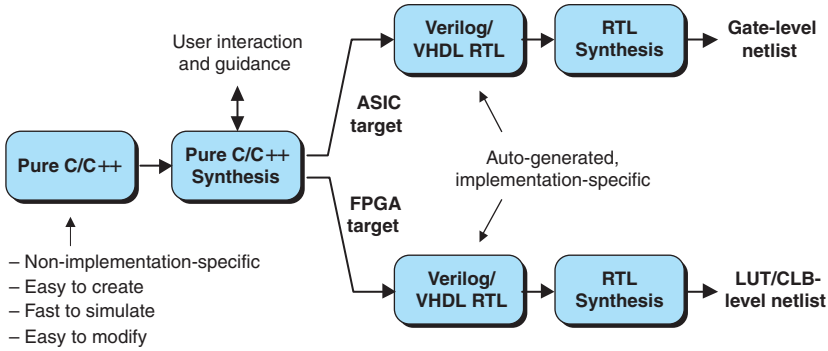


FIGURE 6-7 A pure untimed C/C++-based design flow.

representation. In fact, pure C/C++ models are already widely created and used by system designers for algorithm and system validation.

- *Evaluating alternative implementations is fast and efficient*: Modifying and reverifying pure untimed C/C++ to perform a series of what-if evaluations of alternative microarchitecture implementations is fast and efficient. This facilitates the design team's ability to arrive at fundamentally superior microarchitecture solutions. In turn, this can result in significantly smaller and faster designs as compared to flows based on traditional hand-coded RTL methods.
- *Accommodating specification changes is relatively easy*: If any changes to the specification are made during the course of the project, it's relatively easy to implement and evaluate these changes in a pure untimed C/C++ representation, thereby allowing the changes to be folded into the resulting implementation.

Furthermore, as noted earlier in this chapter, one of the most significant problems associated with existing SystemC and augmented C/C++-based design flows is that the implementation intelligence associated with the design has to be hard-coded into the model, which therefore becomes implementation specific.

A key aspect associated with a pure untimed C/C++-based design flow is that the code presented to the synthesis engine is just what someone would write if he or she didn't have any preconceived hardware implementation or target device architecture in mind. This means that the C/C++ code that system designers write today is an ideal input to this form of synthesis. The only modification typically required to use a pure C/C++ model with the synthesis engine is to add a single special comment to the source code to indicate the top of the functional portion of the design (anything conceptually above this point is considered to form part of the testbench).

As opposed to adding intelligence to the source code (thereby locking it into a target implementation), all of the intelligence is provided by the user controlling and guiding the synthesis engine itself (Figure 6-7).

Once the synthesis engine has parsed the source code, the user can use it to perform microarchitecture trade-offs and evaluate their effects in terms of size and speed. The synthesis engine analyzes the code, identifies its various constructs and operators, along with their associated data and memory dependencies, and automatically provides for parallelism wherever possible. The engine also provides a graphical interface that allows the user to specify how different elements should be handled. For example, the interface

- allows the user to associate ports with registers or RAM blocks;
- identifies constructs like loops and allows the user to specify on an individual basis whether they should be fully unraveled, partially unraveled, or left alone;
- allows the user to specify whether loops and other constructs should be pipelined;
- allows the user to perform resource sharing on specific entities;
- and so forth.

These evaluations are performed on the fly, and the synthesis engine reports total size/area and latency in terms of clock cycles and I/O delays (or throughput time/cycles in the case of pipelined designs). The user-defined configuration associated with each what-if scenario can be named, saved, and reused as required (it would be almost impossible to perform these trade-offs in a timely manner using a conventional hand-coded RTL-based flow).

Key Concept

The fact that the pure untimed C/C++ source code used by the synthesis engine is not required to contain any implementation intelligence and that all such intelligence is supplied by controlling the engine itself means that the same source code can be easily retargeted to alternative microarchitectures and different implementation technologies.

Once the user's evaluations are completed, clicking the "Go" button causes the synthesis engine to generate corresponding RTL VHDL. This code can subsequently be used by conventional logic synthesis or physically aware synthesis applications to generate the netlist used to drive the downstream implementation (place-and-route, etc.) tools.

FAQs

Why not synthesize directly into a gate-level netlist?

As usual, it would be possible to synthesize the pure untimed C/C++ directly into a gate-level netlist (this alternative is not shown in Figure 6-7). However, generating the intermediate RTL provides a comfort zone for the engineers by allowing them to check that they are satisfied with the implementation decisions that have been

made during the course of the C/C++ to RTL translation. Furthermore, generating intermediate RTL is useful because this is the level of abstraction where hardware design engineers generally stitch together the various functional blocks forming their designs. Large portions of today's designs are typically presented in the form of IP blocks represented in RTL. This means that the intermediate RTL step shown in Figure 6-7 is a useful point in the design flow for integrating and verifying the entire hardware system. The design engineers can then take full advantage of their existing RTL synthesis technology, which is mature, robust, and well understood.

Different Levels of Synthesis Abstraction

The fundamental difference between the various C/C++-based flows presented in this chapter is the level of synthesis abstraction each can support. For example, although SystemC offers significant system-level, algorithmic, and transaction-level modeling capabilities, its synthesizable subset is at a relatively low level of abstraction. Similarly, although augmented C/C++ representations are closer to pure C/C++ than are their SystemC counterparts, which means that they simulate much more quickly, their synthesizable subset remains significantly lower than would be ideal.

This lack of synthesis abstraction causes the timed SystemC and augmented C/C++ representations to be implementation specific. In turn, this makes them difficult to create and modify and significantly reduces their flexibility with regard to performing what-if evaluations and retargeting them toward alternative implementation technologies (Figure 6-8).

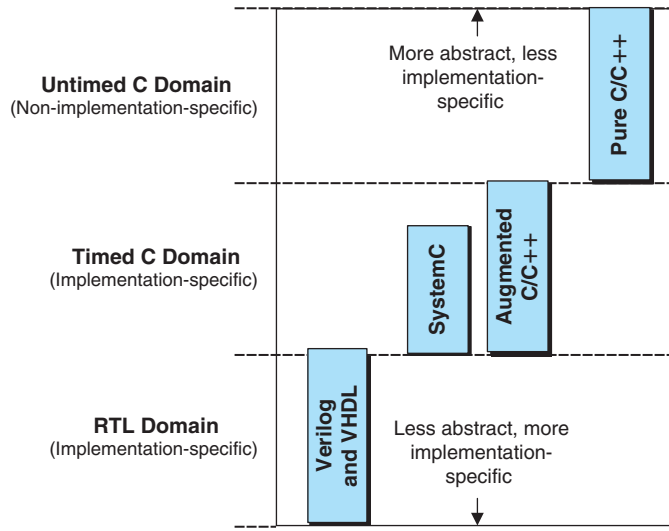


FIGURE 6-8 Different levels of C/C++ synthesis abstraction.

By comparison, the latest generation of pure untimed C/C++ synthesis technology supports a high level of synthesis abstraction. Non-implementation-specific C/C++ models are very compact and can be quickly and easily created and modified. By means of the synthesis engine itself, the user can quickly and easily perform what-if evaluations and retarget the design toward alternative implementation technologies. The result is that a pure C/C++-based design flow can dramatically speed implementation and increase design flexibility as compared to other C/C++-based flows.

Insider Info

Before anyone starts to pen irate letters claiming the author is anti-SystemC, it should be reiterated that the discussions presented here are focused on the use of the various flavors of C/C++ in the context of FPGA implementation flows. In this case, the tool-chain used to progress SystemC representations through to actual implementations is relatively immature and unsophisticated.

When it comes to system-level modeling and verification applications, however, SystemC can be extremely efficacious (many users see SystemC and SystemVerilog being used in conjunction with each other, with SystemC being employed for the initial system-level design representation, and then SystemVerilog being used to "flesh out" the implementation-level details.

Mixed-language Design and Verification Environments

Last, but not least, we should note that a number of EDA companies can provide mixed-level design and verification environments that can support the cosimulation of models specified at multiple levels of abstraction.

In some cases, this may simply involve linking a C/C++ model to a Verilog simulator via its *programming language interface* (PLI) or to a VHDL simulator via its *foreign language interface* (FLI). Alternatively, one might find a SystemC environment with the capability to accept blocks represented in Verilog or VHDL.

And then there are very sophisticated environments that start with a graphical block-based editor showing the design's major functional units, where the contents of each block can be represented using the following:

- VHDL
- Verilog
- SystemVerilog
- SystemC
- Handel-C
- Pure C/C++

The top-level design might be in a traditional HDL that calls submodules in the various HDLs and in one or more flavors of C/C++. Alternatively, the

top-level design might be in one of the flavors of C/C++ that calls submodules in the various languages.

In this type of environment, the VHDL, Verilog, and SystemVerilog representations are usually handled by a single-kernel simulation engine. This engine is then cosimulated with appropriate engines for the various flavors of C/C++. Furthermore, this type of environment will incorporate source-code debuggers that support the various flavors of C/C++; it will allow testbenches to be created using any of the languages; and supporting tools like graphical waveform displays will be capable of displaying signals and variables associated with any of the language blocks.

In reality, the various mixed-language design and verification environment solution combinations and permutations change on an almost weekly basis, so you need to take a good look at what's out there before you leap into the fray.

Key Concept

One advantage of a mixed-language design and verification environment is that you can continue to use your original C/C++ testbench to drive the downstream version of your design in VHDL/Verilog at the RTL and gate levels of abstraction. You may need to tweak a few things, but that's much better than rewriting everything from the ground up.

DSP-BASED DESIGN FLOWS

Digital signal processing includes compression, decompression, modulation, error correction, filtering, and otherwise manipulating audio (voice, music, etc.), video, image, and similar data for such applications as telecommunications, radar, and image processing (including medical imaging). In many cases, the data to be processed starts out as a signal in the real (analog) world. This analog signal is periodically sampled, with each sample being converted into a digital equivalent by means of an analog-to-digital (A/D) converter (Figure 6-9).

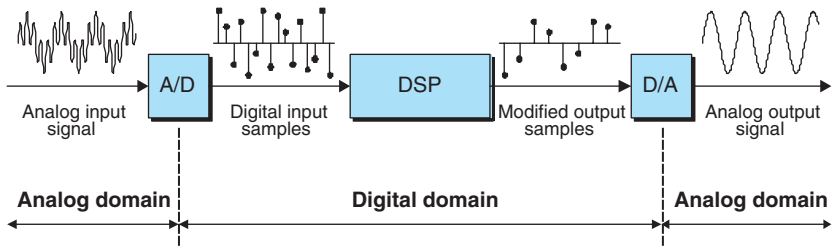


FIGURE 6-9 What is DSP?

These samples are then processed in the digital domain. In many cases, the processed digital samples are subsequently converted into an analog equivalent by means of a digital-to-analog (D/A) converter.

DSP occurs all over the place—in cell phones and telephone systems; CD, DVD, and MP3 players; cable desktop boxes; wireless and medical equipment; electronic vision systems; … the list goes on. This means that the overall DSP market is huge.

Alternative DSP Implementations

As usual, nothing is simple because DSP tasks can be implemented in a number of different ways:

- *A general-purpose microprocessor* (μP): This may also be referred to as a central processing unit (CPU) or a microprocessor unit (MPU). The processor can perform DSP by running an appropriate DSP algorithm.
- *A digital signal processor* (DSP): This is a special form of microprocessor chip (or core, as discussed below) that has been designed to perform DSP tasks much faster and more efficiently than can be achieved by means of a general-purpose microprocessor.
- *Dedicated ASIC hardware*: For the purposes of these discussions, we will assume that this refers to a custom hardware implementation that executes the DSP task. However, we should also note that the DSP task could be implemented in software by including a microprocessor or DSP core on the ASIC.
- *Dedicated FPGA hardware*: For the purposes of these discussions, we will assume that this refers to a custom hardware implementation that executes the DSP task. Once again, however, we should also note that the DSP functionality could be implemented in software by means of an embedded microprocessor core on the FPGA.

System-level Evaluation and Algorithmic Verification

Irrespective of the final implementation technology (μP, DSP, ASIC, FPGA), if one is creating a product that is to be based on a new DSP algorithm, it is common practice to first perform system-level evaluation and algorithmic verification using an appropriate environment (we consider this in more detail later in this chapter).

Although this book attempts to avoid focusing on companies and products as far as possible, it is encumbant on us to mention that—at the time of this writing—the de facto industry standard for DSP algorithmic verification is MATLAB® from The MathWorks (www.mathworks.com).2

For the purposes of these discussions, therefore, we shall refer to MATLAB as necessary. However, it should be noted that there are a number of other very powerful tools and environments available to DSP developers. For example,

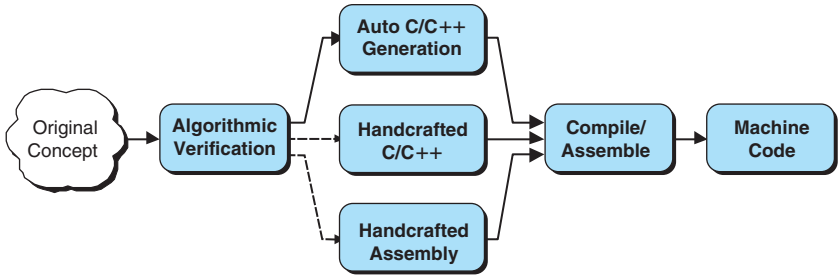


FIGURE 6-10 A simple design flow for a software DSP realization.

Simulink® from The MathWorks has a certain following; the Signal Processing Worksystem (SPW) environment from CoWare3 (www.coware.com) is very popular, especially in telecom markets; and tools from Elanix (www.elanix.com) also find favor with many designers.

Software Running on a DSP Core

Let's assume that our new DSP algorithm is to be implemented using a microprocessor or DSP chip (or core). In this case, the flow might be as shown in Figure 6-10.

- The process commences with someone having an idea for a new algorithm or suite of algorithms. This new concept typically undergoes verification using tools such as MATLAB as discussed above. In some cases, one might leap directly from the concept into handcrafting C/C++ (or assembly language).
- Once the algorithms have been verified, they have to be regenerated in C/C++ or in assembly language. MATLAB can be used to generate C/C++ tuned for the target DSP core automatically, but in some cases, design teams may prefer to perform this translation step by hand because they feel that they can achieve a more optimal representation this way. As yet another alternative, one might first auto-generate C/C++ code from the algorithmic verification environment, analyze and profile this code to determine any performance bottlenecks, and then recode the most critical portions by hand.
- Once you have your C/C++ (or assembly language) representation, you compile it (or assemble it) into the machine code that will ultimately be executed by the microprocessor or DSP core.

This type of implementation is very flexible because any desired changes can be addressed relatively quickly and easily by simply modifying and recompiling the source code. However, this also results in the slowest performance for the DSP algorithm because microprocessor and DSP chips are both classed

as Turing machines. This means that their primary role in life is to process instructions, so both of these devices operate as follows:

- Fetch an instruction.
- Decode the instruction.
- Fetch a piece of data.
- Perform an operation on the data.
- Store the result somewhere.
- :
- Fetch another instruction and start all over again.

Key Concept

Of course, the DSP algorithm actually runs on hardware in the form of the microprocessor or DSP, but we consider this to be a software implementation because the actual (physical) manifestation of the algorithm is the program that is executed on the chip.

Dedicated DSP Hardware

There are myriad ways in which one might implement a DSP algorithm in an ASIC or FPGA—the latter option being the focus of this chapter, of course. But before we hurl ourselves into the mire, let's first consider how different architectures can affect the speed and area (in terms of silicon real estate) of the implementation.

DSP algorithms typically require huge numbers of multiplications and additions. As a really simple example, let's assume that we have a new DSP algorithm that contains an expression something like the following:

```
Y = (A * B) + (C * D) + (E * F) + (G * H);
```

As usual, this is a generic syntax that does not favor any particular HDL and is used only for the purposes of these discussions. Of course, this would be a minuscule element in a horrendously complex algorithm, but DSP algorithms tend to contain a lot of this type of thing.

The point is that we can exploit the parallelism inherent in hardware to perform DSP functions much more quickly than can be achieved by means of software running on a DSP core. For example, suppose that all of the multiplications were performed in parallel (simultaneously) followed by two stages of additions (Figure 6-11).

Remembering that multipliers are relatively large and complex and that adders are sort of large, this implementation will be very fast, but will consume a correspondingly large amount of chip resources.

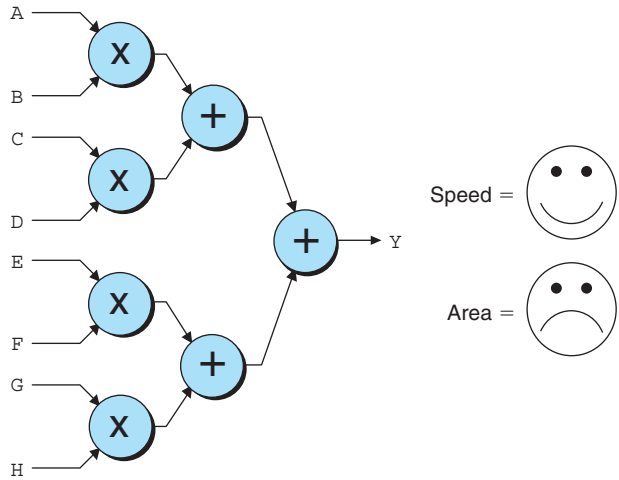


FIGURE 6-11 A parallel implementation of the function.

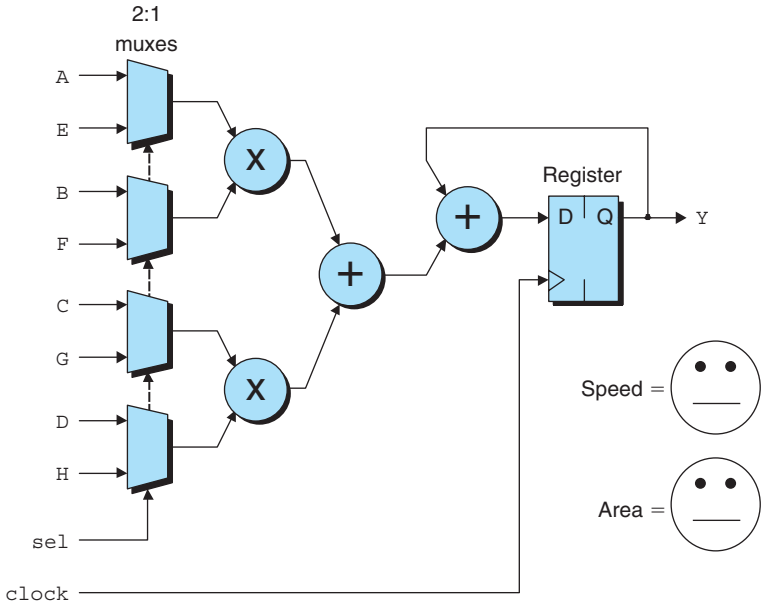


FIGURE 6-12 An in-between implementation of the function.

As an alternative, we might employ *resource sharing* (sharing some of the multipliers and adders between multiple operations) and opt for a solution that is a mixture of parallel and serial (Figure 6-12).

This solution requires the addition of four 2:1 multiplexers and a register (remember that each of these will be the same multibit width as their respective

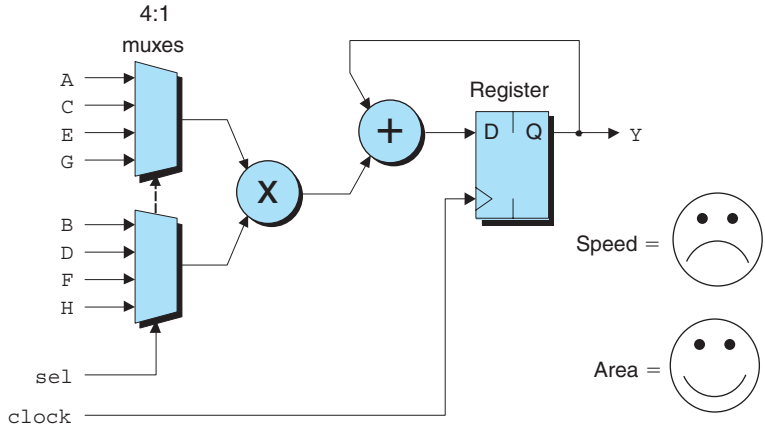


FIGURE 6-13 A serial implementation of the function.

signal paths). However, multiplexers and registers consume much less area than the two multipliers and adder that are no longer required as compared to our initial solution.

On the downside, this approach is slower, because we must first perform the (A * B) and (C * D) multiplications, add the results together, add this total to the existing contents of the register (which will have been initialized to contain zero), and store the result in the register. Next, we must perform the (E * F) and (G * H) multiplications, add these results together, add this total to the existing contents of the register (which currently contains the results from the first set of multiplications and additions), and store this result in the register.

As yet another alternative, we might decide to use a fully serial solution (Figure 6-13).

This latter implementation is very efficient in terms of area because it requires only a single multiplier and a single adder. This is the slowest implementation, however, because we must first perform the (A * B) multiplication, add the result to the existing contents of the register (which will have been initialized to contain zero), and store the total in the register. Next, we must perform the (C * D) multiplication, add this result to the existing contents of the register, and store this new total in the register. And so forth for the remaining multiplication operations. (Note that when we say "this is the slowest implementation," we are referring to these hardware solutions, but even the slowest hardware implementation remains much, much faster than a software equivalent running on a microprocessor or DSP.)

DSP-related Embedded FPGA Resources

As previously discussed, some functions like multipliers are inherently slow if they are implemented by connecting a large number of programmable logic

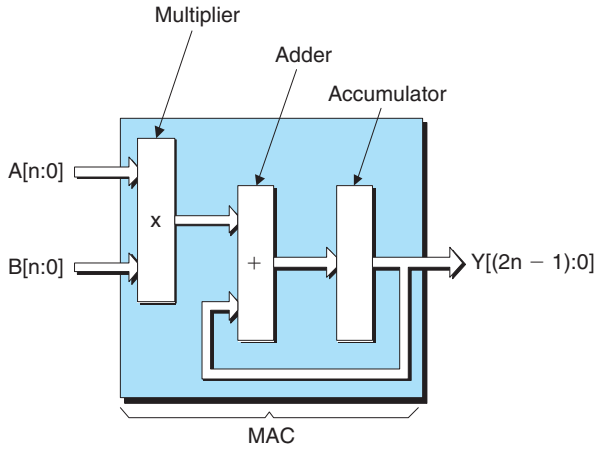


FIGURE 6-14 The functions forming a MAC.

blocks together inside an FPGA. Since many applications require these functions, many FPGAs incorporate special hard-wired multiplier blocks. (These are typically located in close proximity to embedded RAM blocks because these functions are often used in conjunction with each other.)

Similarly, some FPGAs offer dedicated adder blocks. One operation that is very common in DSP-type applications is ulate. As its name would suggest, this function multiplies two numbers together and adds the result into a running total stored in an accumulator (register). Hence, it is commonly referred to as a MAC, which stands for multiply, add, and accumulate (Figure 6-14).

Note that the multiplier, adder, and register portions of the serial implementation of our function shown in Figure 6-13 offer a classic example of a MAC. If the FPGA you are working with supplies only embedded multipliers, you would be obliged to implement this function by combining the multiplier with an adder formed from a number of programmable logic blocks, while the result would be stored in a block RAM or in a number of distributed RAMs. Life becomes a little easier if the FPGA also provides embedded adders, and some FPGAs provide entire MACs as embedded functions.

FPGA-centric Design Flows for DSPs

At the time of this writing, using FPGAs to perform DSP is still relatively new. Thus, there really are no definitive design flows or methodologies here—everyone seems to have his or her unique way of doing things, and whichever option you choose, you'll almost certainly end up breaking new ground one way or another.

Domain-specific Languages

The way of the world is that electronic designs increase in size and complexity over time. To manage this problem while maintaining—or, more usually,

increasing—productivity, it is necessary to keep raising the level of abstraction used to capture the design's functionality and verify its intent.

For this reason the gate-level schematics were superceded by the RTL representations in VHDL and Verilog, as discussed in Chapter 5. Similarly, the drive toward C-based flows as discussed earlier is powered by the desire to capture complex concepts quickly and easily while facilitating architectural analysis and exploration.

In the case of specialist areas such as DSPs, system architects and design engineers can achieve a dramatic improvement in productivity by means of domain-specific languages (DSLs), which provide more concise ways of representing specific tasks than do general-purpose languages such as C/C++ and SystemC.

One such language is MATLAB, which allows DSP designers to represent a signal transformation, such as an FFT, that can potentially take up an entire FPGA, using a single line of code4 along the lines of

```
y = fft(x);
```

Actually, the term MATLAB refers both to a language and an algorithmic-level simulation environment. To avoid confusion, it is common to talk about M-code (meaning "MATLAB code") and M-files (files containing MATLAB code).

Insider Info

Some engineers in the trenches occasionally refer to the "M language," but this is not argot favored by the folks at The MathWorks.

In addition to sophisticated transformation operators like the FFT shown above, there are also much simpler transformations like adders, subtractors, multipliers, logical operators, matrix arithmetic, and so forth. The more complex transformations like an FFT can be formed from these fundamental entities if required. The output from each transformation can be used as the input to one or more downstream transformations, and so forth, until the entire system has been represented at this high level of abstraction.

One important point is that such a system-level representation does not initially imply a hardware or software implementation. In the case of DSP core, for example, it could be that the entire function is implemented in software as discussed earlier in this chapter. Alternatively, the system architects could partition the design such that some functions are implemented in software, while other performance-critical tasks are implemented in hardware using dedicated ASIC or FPGA fabric. In this case, one typically needs to have access to a hardware or software codesign environment. For the purposes of these discussions, however, we shall assume pure hardware implementations.

Key Concept

M-files can contain scripts (actions to be performed) or transformations or a mixture of both. Also, M-files can call other M-files in a hierarchical manner. The primary (top-level) M-file typically contains a script that defines the simulation run. This script might prompt the user for information like the values of filter coefficients that are to be used, the name of an input stimulus file, and so forth, and then call other M-files and pass them these user-defined values as required.

System-level Design and Simulation Environments

System-level design and simulation environments are conceptually at a higher level than DSLs. One well-known example of this genre is Simulink from The MathWorks. Depending on who you're talking to, there may be a perception that Simulink is simply a graphical user interface to MATLAB. In reality, however, it is an independent dynamic modeling application that works *with* MATLAB.

If you are using Simulink, you typically commence the design process by creating a graphical block diagram of your system showing a schematic of functional blocks and the connections between them. Each of these blocks may be user-num defined, or they may originate in one of the libraries supplied with Simulink (these include DSP, communications, and control function block sets). In the case of a user-defined block, you can "push" into that block and represent its contents as a new graphical block diagram. You can also create blocks containing MATLAB functions, M-code, C/C++, FORTRAN … the list goes on.

Once you've captured the design's intent, you use Simulink to simulate and verify its functionality. As with MATLAB, the input stimulus to a Simulink simulation might come from one or more mathematical functions, such as sine-wave generators, or it might be provided in the form of real-world data such as audio or video files. In many cases, it comes as a mixture of both; for example, real-world data might be augmented with pseudorandom noise supplied by a Simulink block.

—Technology Trade-offs—

- The point here is that there's no hard-and-fast rule. Some DSP designers prefer to use MATLAB as their starting point, while others opt for Simulink (this latter case is much rarer in the scheme of things). Some folks say that this preference depends on the user's background (software DSP development versus ASIC/FPGA designs), but others say that this is a load of tosh.

Floating-point versus Fixed-point Representations

Irrespective as to whether one opts for Simulink or MATLAB (or a similar environment from another vendor) as a starting point, the first-pass model

of the system is almost invariably described using floating-point representations. In the context of the decimal number system, this refers to numbers like 1.235 × 103 (that is, a fractional number raised to some power of 10). In the context of applications like MATLAB, equivalent binary values are represented inside the computer using the IEEE standard for double-precision floating-point numbers.

Floating-point numbers of this type have the advantage of providing extremely accurate values across a tremendous dynamic range. However, implementing floating-point calculations of this type in dedicated FPGA or ASIC hardware requires a humongous amount of silicon resources, and the result is painfully slow (in hardware terms). Thus, at some stage, the design will be migrated over to use fixed-point representations, which refers to numbers having a fixed number of bits to represent their integer and fractional portions. This process is commonly referred to as *quantization*.

This is totally system/algorithm dependent, and it may take a considerable amount of experimentation to determine the optimum balance between using the fewest number of bits to represent a set of values (thereby decreasing the amount of silicon resources required and speeding the calculations), while maintaining sufficient accuracy to perform the task in hand. (One can think of this trade-off in terms of how much noise the designer is willing to accept for a given number of bits.) In some cases, designers may spend days deciding "should we use 14, 15, or 16 bits to represent these particular values?" And, just to increase the fun, it may be best to vary the number of bits used to represent values at different locations in the system/algorithm.

Things start to get really fun in that the conversion from floating-point to fixed-point representations may take place upstream in the system/algorithmic design and verification environment, or downstream in the C/C++ code. This is shown in more detail in the "System/algorithmic level to C/C++" section below. Suffice it to say that if one is working in a MATLAB environment, these conversions can be performed by passing the floating-point signals through special transformation functions called *quantizers*. Alternatively, if one is working in a Simulink environment, the conversions can be performed by running the floating-point signals through special fixed-point blocks.

System/algorithmic Level to RTL (Manual Translation)

At the time of this writing, many DSP design teams commence by performing their system-level evaluations and algorithmic validation in MATLAB (or the equivalent) using floating-point representations. (It is also very common to include an intermediate step in which a fixed-point C/C++ model is created for use in rapid simulation/validation.) At this point, many design teams bounce directly into hand-coding fixed-point RTL equivalents of the design in VHDL or Verilog (Figure 6-14a). Alternatively, they may first transition the floating-point representations into their fixed-point counterparts at the system/algorithmic level, and then hand-code the RTL in VHDL or Verilog (Figure 6-14b).

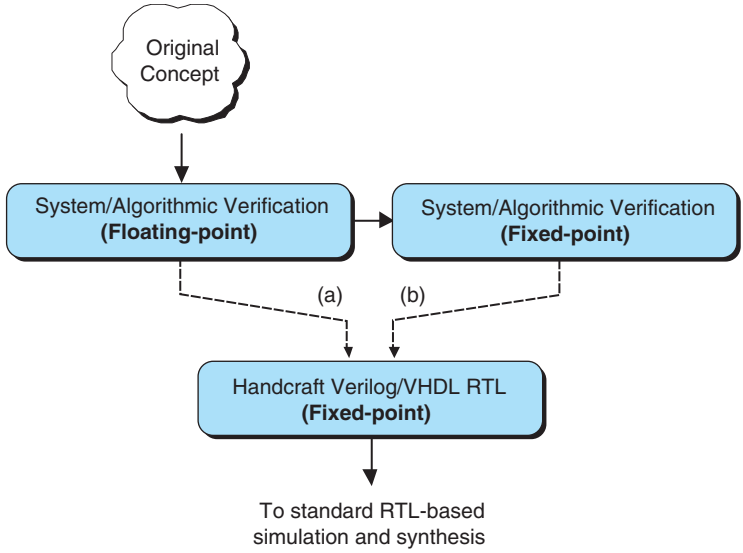


FIGURE 6-14a,b Manual RTL generation.

Of course, once an RTL representation of the design has been created, we can assume the use of the downstream logic-synthesis-based flows that were introduced in Chapter 5.

—Technology Trade-offs—

- There are a number of problems with this flow, not the least being that there is a significant conceptual and representational divide between the system architects working at the system/algorithmic level and the hardware design engineers working with RTL representations in VHDL or Verilog.
- Because the system/algorithmic and RTL domains are so different, manual translation from one to the other is time-consuming and prone to error.
- There is also the fact that the resulting RTL is implementation specific because realizing the optimal design in an FPGA requires a different RTL coding style from that used for an optimal ASIC implementation.
- Another consideration is that manually modifying and reverifying RTL to perform a series of what-if evaluations of alternative microarchitecture implementations is extremely time-consuming (such evaluations may include performing certain operations in parallel versus sequential, pipelining portions of the design versus nonpipelining, sharing common resources—for example, two operations sharing a single multiplier—versus using dedicated resources, etc.)
- Similarly, if any changes are made to the original specification during the course of the project, it's relatively easy to implement and evaluate these changes in the system-/algorithmic-level representations, but

subsequently folding these changes into the RTL by hand can be painful and time-consuming.

System/Algorithmic Level to RTL (Automatic-generation)

As was noted in the previous section, performing system-/algorithmic-level-to-RTL translation manually is time-consuming and prone to error. There are alternatives, however, because some system-/algorithmic-level design environments offer direct VHDL or Verilog RTL code generation (Figure 6-15).

As usual, the system-/algorithmic-level design would commence by using floating-point representations. In one version of the flow, the system/algorithmic environment is used to migrate these representations into their fixed-point counter-parts and then to generate the equivalent RTL in VHDL or Verilog automatically (Figure 6-15a).

Alternatively, a third-party environment might be used to take the floating-point system-/algorithmic-level representation, autointeractively quantize it into its fixed-point counterpart, and then automatically generate the equivalent RTL in VHDL or Verilog (Figure 6-15b).

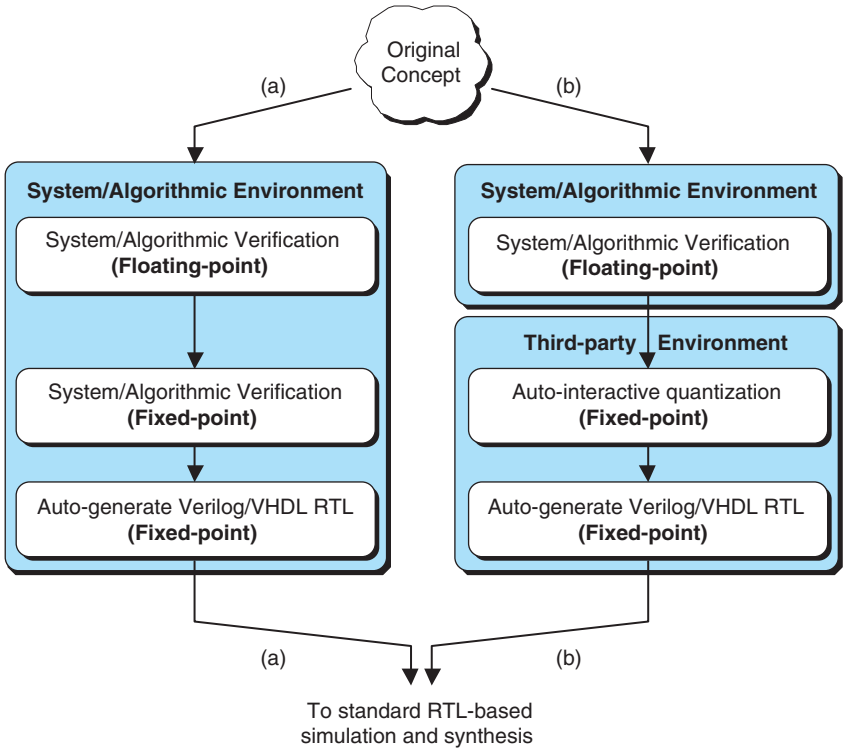


FIGURE 6-15 Direct RTL generation.

As before, once an RTL representation of the design has been created, we can assume the use of the downstream logic-synthesis-based flows that were introduced in Chapter 5.

System/Algorithmic Level to C/C++

Due to the problems associated with exploring the design at the RTL level, there is an increasing trend to use a stepping-stone approach. This involves transitioning from the system-/algorithmic-level domain into to some sort of C/C++ representation, which itself is subsequently migrated into an RTL equivalent. One reason this is attractive is that the majority of DSP design teams already generate a C/C++ model for use as a golden (reference) model, in which case this sort of comes for free as far as the downstream RTL design engineer is concerned.

Of course, the first thing to decide is when and where in the flow one should transition from floating-point to fixed-point representations (Figure 6-16).

Frighteningly enough, Figure 6-16 shows only a subset of the various potential flows. For example, in the case of the handcrafted options, as opposed to first hand-coding the C/C++ and then gradually transmogrifying this representation into Handel-C or SystemC, one could hand-code directly into these languages.

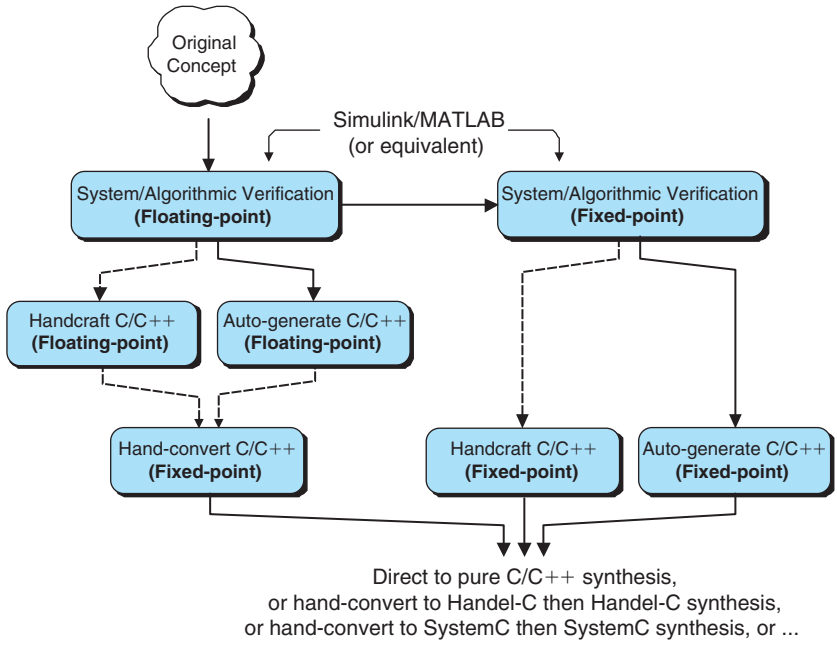


FIGURE 6-16 Migrating from floating point to fixed point.

Key Concept

The main thing to remember is that once we have a fixed-point representation in one of the flavors of C/C++, we can assume the use of the downstream C/C++ flows introduced earlier (one flow of particular interest in this area is the pure untimed C/C++ approach used by Precision C from Mentor).

Block-level IP Environments

Nothing is simple in this world because there is always just one more way to do things. As an example, one might create a library of DSP functional blocks at the system/algorithmic level of abstraction along with a one-to-one equivalent library of blocks at the RTL level of abstraction in VHDL or Verilog.

The idea here is that you could then capture and verify your design using a hierarchy of functional blocks specified at the system/algorithmic level of abstraction. Once you were happy with your design, you could then generate a structural netlist instantiating the RTL-level blocks, and use this to drive downstream simulation and synthesis tools. (These blocks would have to be parameterized at all levels of abstraction to allow you to specify such things as bus widths and so forth.)

As an alternative, the larger FPGA vendors typically offer IP core generators (in this context, the term core is considered to refer to a block that performs a specific logical function; it does not refer to a microprocessor or DSP core). In several cases, these core generators have been integrated into system-/algorithmic-level environments. This means that you can create a design based on a collection of these blocks in the system-/algorithmic-level environment, specify any parameters associated with these blocks, and perform your system-/algorithmic-level verification.

Later, when you're ready to rock and roll, the core generator will automatically generate the hardware models corresponding to each of these blocks. (The system-/algorithmic-level models and the hardware models ensuing from the core generator are bit identical and cycle identical.) In some cases the hardware blocks will be generated as synthesizable RTL in VHDL or Verilog. Alternatively, they may be presented as firm cores at the LUT/CLB level of abstraction, thereby making the maximum use of the targeted FPGA's internal resources.

—Technology Trade-offs—

- One big drawback associated with this approach is that, by their very nature, IP blocks are based on hard-coded microarchitectures. This means that the ability to create highly tuned implementations to address specific design goals is somewhat diminished. The result is that IP-based flows may achieve an implementation faster with less risk, but such an implementation

may be less optimal in terms of area, performance, and power as compared to a custom hardware implementation.

Don't Forget the Testbench!

One point the folks selling you DSP design tools often neglect to mention is the test bench. For example, let's assume that your flow involves taking your system-/algorithmic-level design and hand-translating it into RTL. In that case, you are going to have to do the same with your testbench. In many cases, this is a nontrivial task that can take days or weeks!

Or let's say that your flow is based on taking your floating- point system-/algorithmic-level design and hand-translating it into floating-point C/C++, at which point you will wish to verify this new representation. Then you might take your floating-point C/C++ and hand-translate it into fixed-point C/C++, at which point you will wish to verify this representation. And then you might take your fixed-point C/C++ and (hopefully) automatically synthesize an equivalent RTL representation, at which point … but you get my drift.

The problem is that at each stage you are going to have to do the same thing with your testbench (unless you do something cunning as discussed in the next (and last—hurray!) section.

Mixed DSP and VHDL/Verilog etc. Environments

In the previous chapter, we noted that a number of EDA companies can provide mixed-level design and verification environments that can support the cosimulation of models specified at multiple levels of abstraction. For example, one might start with a graphical block-based editor showing the design's major functional units, where the contents of each block can be represented using

- VHDL
- Verilog
- SystemVerilog
- SystemC
- Handel-C
- Pure C/C++

In this case, the top-level design might be in a traditional HDL that calls submodules represented in the various HDLs and in one or more flavors of C/C++. Alternatively, the top-level design might be in one of the flavors of C/C++ that calls submodules in the other languages.

More recently, integrations between system-/algorithmic-level and implementation-level environments have become available. The way in which this works depends on who is doing what and what that person is trying to. For example, a system architect working at the system/algorithmic level (e.g., in MATLAB) might decide to replace one or more blocks with equivalent representations in VHDL or Verilog at the RTL level of abstraction. Alternatively,

a design engineer working in VHDL or Verilog at the RTL level of abstraction might decide to call one or more blocks at the system/algorithmic level of abstraction.

Both of these cases require cosimulation between the system-/algorithmic-level environment and the VHDL/Verilog environment, the main difference being who calls whom. Of course, this sounds easy if you say it quickly, but there is a whole host of considerations to be addressed, such as synchronizing the concept of time between the two domains and specifying how different signal types are translated as they pass from one domain to the other (and back again).

Insider Info

Treat any canned demonstration with a healthy amount of suspicion. If you are planning on doing this sort of thing, you need to sit down with the vendor's engineer and work your own example through from beginning to end. Call me an old cynic if you will, but my advice is to let their engineer guide you, while keeping your hands firmly on the keyboard and mouse. (You'd be amazed how much activity can go on in just a few seconds should you turn your head in response to the age-old question, "Good grief! Did you see what just flew by the window?")

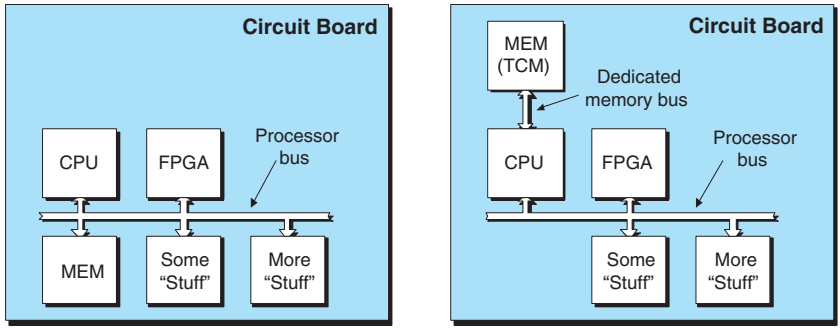
EMBEDDED PROCESSOR-BASED DESIGN FLOWS

We are concerned only with electronic systems that include one or more FPGAs on the printed circuit board (PCB). The vast majority of such systems also make use of a general-purpose microprocessor, or μP, to perform a variety of control and data-processing applications. This is often referred to as the central processing unit (CPU) or microprocessor unit (MPU).

Until recently, the CPU and its peripherals typically appeared in the form of discrete chips on the circuit board. There are an almost infinite number of possible scenarios here, but the two main ones involve the way in which the CPU is connected to its memory (Figure 6-17).

In both of these scenarios, the CPU is connected to an FPGA and some other stuff via a general-purpose processor bus. (By "stuff" we predominantly mean peripheral devices such as counter timers, interrupt controllers, communications devices, etc.)

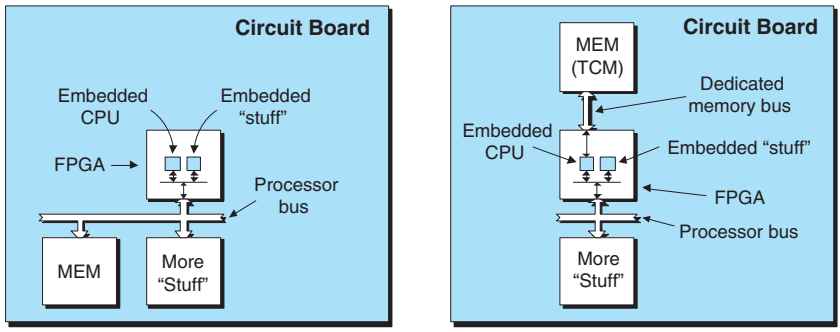
In some cases, the main memory (MEM) will also be connected to the CPU by means of the main processor bus, as shown in Figure 6-17a (actually, this connection will be via a special peripheral called a memory controller, which is not shown here because we're trying to keep things simple). Alternatively, the memory may be connected directly to the CPU by means of a dedicated memory bus, as shown in Figure 6-17b.



(a) Memory connected to CPU via general-purpose processor bus

(b) Tightly coupled memory (TCM) connected to CPU via dedicated bus

FIGURE 6-17 Two scenarios at the circuit board level



(a) Memory connected to CPU via general-purpose processor bus

(b) Tightly coupled memory (TCM) connected to CPU via dedicated bus

FIGURE 6-18 Two scenarios at the FPGA level.

The point is that presenting the CPU and its various peripheral devices in the form of dedicated chips on the circuit board costs money and occupies real estate. It also impacts the reliability of the board because every solder joint (connection point) is a potential failure mechanism.

One alternative is to embed the CPU along with some of its peripherals in the FPGA itself (Figure 6-18).

It is common for a relatively small amount of memory used by the CPU to be included locally in the FPGA. At the time of this writing, however, it is rare for all of the CPU's memory to be included in the FPGA.

Creating an FPGA design of this type brings a whole slew of new problems to the table:

- First, the system architects have to decide which functions will be implemented in software (as instructions to be executed by the CPU) and which functions will be implemented in hardware (using the main FPGA fabric).

- Next, the design environment must support the concept of coverification, in which the hardware and embedded software portions of the system can be verified together to ensure that everything works as it should.

Both of these topics are considered in more detail later in this chapter.

Hard versus Soft Cores

Hard Cores

As defined previously, a hard microprocessor core is one that is implemented as a dedicated, predefined (hardwired) block (these cores are only available in certain device families). Each of the main FPGA vendors has opted for a particular processor type to implement its hard cores. For example, Altera offers embedded ARM processors, QuickLogic has opted for MIPS-based solutions, and Xilinx sports PowerPC cores.

Of course, each vendor will be delighted to explain at great length why its implementation is far superior to any of the others (the problem of deciding which one actually is better is only compounded by the fact that different processors may be better suited to different tasks).

As noted in Chapter 2, there are two main approaches for integrating such cores into the FPGA. The first is to **locate it in a strip to the side of the main FPGA fabric** (Figure 6-19).

In this scenario, all of the components are typically formed on the same silicon chip, although they could also be formed on two chips and packaged as a *multichip module* (MCM).

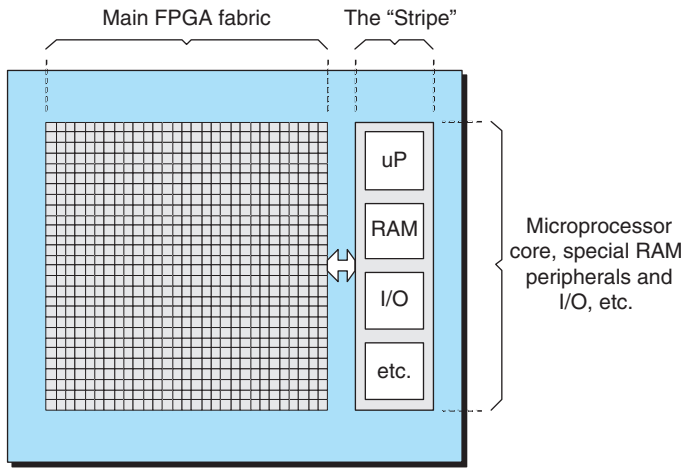


FIGURE 6-19 Bird's-eye view of chip with embedded core outside of the main fabric.

One advantage of this implementation is that the main FPGA fabric is identical for devices with and without the embedded microprocessor core, which can make things easier for the design tools used by the engineers. The other advantage is that the FPGA vendor can bundle a whole load of additional functions in the strip to complement the microprocessor core, such as memory and special peripherals.

The second alternative is to **embed one or more microprocessor cores directly into the main FPGA fabric**. One, two, and even four core implementations are currently available at the time of this writing (Figure 6-20).

In this case, the design tools have to be able to take account of the presence of these blocks in the fabric; any memory used by the core is formed from embedded RAM blocks, and any peripheral functions are formed from groups of general-purpose programmable logic blocks. Proponents of this scheme can argue that there are inherent speed advantages to be gained from having the microprocessor core in intimate proximity to the main FPGA fabric.

Soft Microprocessor Cores

As opposed to embedding a microprocessor physically into the fabric of the chip, it is possible to configure a group of programmable logic blocks to act as a microprocessor. These are typically called "soft cores," but they may be more precisely categorized as either soft or firm, depending on the way in which the microprocessor's functionality is mapped onto the logic blocks. For example, if the core is provided in the form of an RTL netlist that will be synthesized with the other logic, then this truly is a soft implementation. Alternatively, if the core is presented in the form of a placed and routed block of LUTs/CLBs, then this would typically be considered a firm implementation.

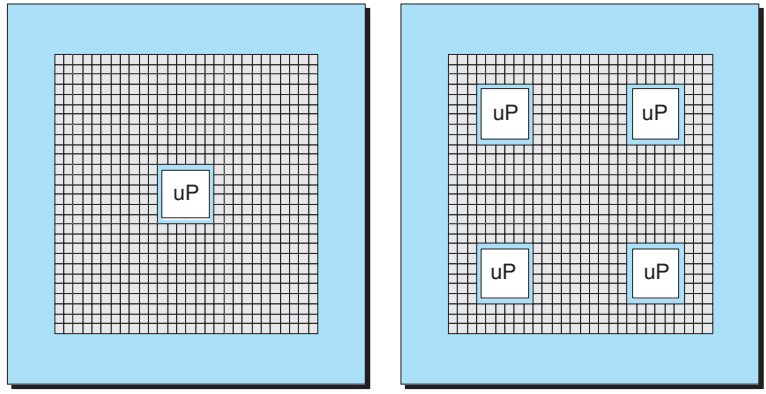


FIGURE 6-20 Bird's-eye view of chips with embedded cores inside the main fabric.

In both of these cases, all of the peripheral devices like counter timers, interrupt controllers, memory controllers, communications functions, and so forth are also implemented as soft or firm cores (the FPGA vendors are typically able to supply a large library of such cores).

—Technology Trade-offs—

- Soft cores are slower and simpler than their hard-core counterparts (of course they are still incredibly fast in human terms). However, in addition to being practically free, they also have the advantages that you only have to implement a core if you need it and that you can instantiate as many cores as you require until you run out of resources in the form of programmable logic blocks.

Once again, each of the main FPGA vendors has opted for a particular processor type to implement its soft cores. For example, Altera offers the Nios, while Xilinx sports the MicroBlaze. The Nios has both 16-bit and 32-bit architectural variants, which operate on 16-bit or 32-bit chunks of data, respectively (both variants share the same 16-bit-wide instruction set). By comparison, the MicroBlaze is a true 32-bit machine (that is, it has 32-bit-wide instruction words and performs its magic on 32-bit chunks of data). Once again, each vendor will be more than happy to tell you why its soft core rules and how its competitors' offerings fail to make the grade (sorry, you're on your own here).

One cool thing about the *integrated development environment* (IDE) fielded by Xilinx is that it treats the PowerPC hard core and the MicroBlaze soft core identically. This includes both processors being based on the same CoreConnect processor bus and sharing common soft peripheral IP cores. All of this makes it relatively easy to migrate from one processor to the other.

Also of interest is the fact that Xilinx offers a small 8-bit soft core called the PicoBlaze, which can be implemented using only 150 logic cells (give or take a handful). By comparison, the MicroBlaze requires around 1,000 logic cells (which is still extremely reasonable for a 32-bit processor implementation, especially when one is playing with FPGAs that can contain 70,000 or more such cells).

Insider Info

Some cynics say that those aspects of a design that are well understood are implemented in hardware, while any portions of the design that are somewhat undefined at the beginning of the design process are often relegated to a software realization (on the basis that the software can be tweaked right up until the last minute).

Partitioning a Design into Its Hardware and Software Components

As noted in Chapter 2, almost any portion of an electronic design can be realized in hardware (using logic gates and registers, etc.) or software (as instructions to be executed on a microprocessor). One of the main partitioning criteria is how fast you wish the various functions to perform their tasks:

- *Picosecond and nanosecond logic*: This has to run insanely fast, which mandates that it be implemented in hardware (in the FPGA fabric).
- *Microsecond logic*: This is reasonably fast and can be implemented either in hardware or software (this type of logic is where you spend the bulk of your time deciding which way to go).
- *Millisecond logic*: This is the logic used to implement interfaces such as reading switch positions and flashing light-emitting diodes, or LEDs. It's a pain slowing the hardware down to implement this sort of function (using huge counters to generate delays, for example). Thus, it's often better to implement these tasks as microprocessor code (because processors give you lousy speed—compared to dedicated hardware—but fantastic complexity).

The trick is to solve every problem in the most cost-effective way. Certain functions belong in hardware, others cry out for a software realization, and some functions can go either way depending on how you feel you can best use the resources (both chip-level resources and hardware/software engineers) available to you.

It is possible to envisage an "ideal" *electronic system level* (ESL) environment in which the system architects initially capture the design via a graphical interface as a collection of functional blocks that are connected together. Each of these blocks could then be provided with a system-/algorithmic level SystemC representation, for example, and the entire design could be verified prior to any decisions being made as to which portions of the design were to be implemented in hardware and software.

When it comes to the partitioning process itself, we might dream of having the ability to tag each graphical block with the mouse and select a hardware or software option for its implementation. All we would then have to do would be to click the "Go" button, and the environment would take care of synthesizing the hardware, compiling the software, and pulling everything together.

And then we return to the real world with a resounding thud. Actually, a number of next-generation design environments show promise, and new tools and techniques are arriving on an almost daily basis. At the time of this writing, however, it is still very common for system architects to partition a design into its hardware and software portions by hand, and to then pass these top-level functions over to the appropriate engineers and hope for the best.

With regard to the software portion of the design, this might be something as simple as a state machine used to control a human-level interface (reading the state of switches and controlling display devices). Although the state machine itself may be quite tricky, this level of software is certainly not rocket science. At the other end of the spectrum, one might have incredibly complex software requirements, including:

- System initialization routines and a hardware abstraction layer
- A hardware diagnostic test suite
- A real-time operating system (RTOS)
- RTOS device drivers
- Any embedded application code

This code will typically be captured in C/C++ and then compiled down to the machine instructions that will be run on the processor core (in extreme cases where one is attempting to squeeze the last drop of performance out of the design, certain routines may be handcrafted in assembly code).

At the same time, the hardware design engineers will typically be capturing their portions of the design at the RTL level of abstraction using VHDL or Verilog (or SystemVerilog).

Today's designs are so complex that their hardware and software portions have to be verified together.

Insider Info

One of the biggest problems to overcome when it comes to the coverification of the hardware and software portions of a design is the two totally different worldviews of their creators. The hardware folks typically visualize their portion of the design as blocks of RTL representing such things as registers, logical functions, and the wires connecting them together. When hardware engineers are debugging their portion of the design, they think in terms of an editor showing their RTL source code, a logic simulator, and a graphical waveform display showing signals changing values at specific times. In a typical hardware design environment, clicking on a particular event in the waveform display will automatically locate the corresponding line of RTL code that caused this event to occur.

By comparison, the software guys and gals think in terms of C/C++ source code, of registers in the CPU (and in the peripherals), and of the contents of various memory locations. When software engineers are debugging a program, they often wish to single-step through the code one line at a time and watch the values in the various registers changing. Or they might wish to set one or more breakpoints (this refers to placing markers at specific points in the code), run the program until they hit one of those breakpoints, and then pause to see what's going on. Alternatively, they might wish to specify certain conditions such as a register containing a particular value, then run the program until this condition is met, and once again pause to see what's happening.

When a software developer is writing application code such as a game, he or she has the luxury of being reasonably confident that the hardware (say, a home computer) is reasonably robust and bug-free. However, it's a different ball game when one is talking about a software engineer creating embedded applications intended to run on hardware that's being designed at the same time. When a problem occurs, it can be mega tricky determining if it was a fault in the software or if the hardware was to blame.

Using an FPGA as Its Own Development Environment

Perhaps the simplest place to start is the scenario where the FPGA is used as its own development environment. The idea here is that you have an SRAM-based FPGA with an embedded processor (hard or soft) mounted on a development board that's connected to your computer. In addition to the FPGA, this development board will also have a memory device that will be used to store the software programs that are to be run by the embedded CPU (Figure 6-21).

Once the system architects have determined which portions of the design are to be implemented in hardware and software, the hardware engineers start to capture their RTL blocks and functions and synthesize them down to a LUT/CLB-level netlist. Meanwhile, the software engineers start to capture their C/C++ programs and routines and compile them down to machine code. Eventually, the LUT/CLB-level netlist will be loaded into the FPGA via a configuration file, the linked machine code image will be loaded into the memory device, and then you let the system run wild and free (Figure 6-22).

Also, any of the machine code that is to be embedded in the FPGA's on-chip RAM blocks would actually be loaded via the configuration file.

Improving Visibility in the Design

The main problem with the scenario discussed in the previous section is lack of "visibility" as to what is happening in the hardware portion of the design.

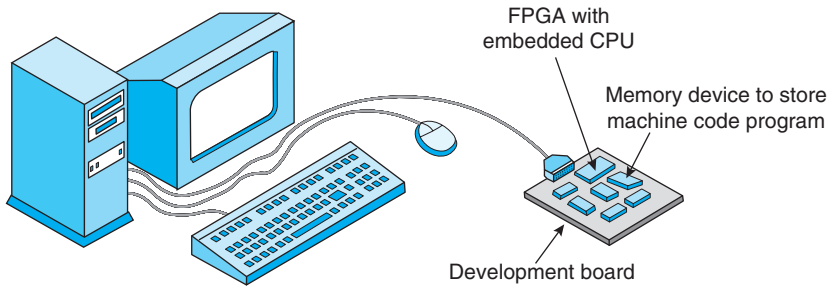


FIGURE 6-21 Using an FPGA as its own development environment.

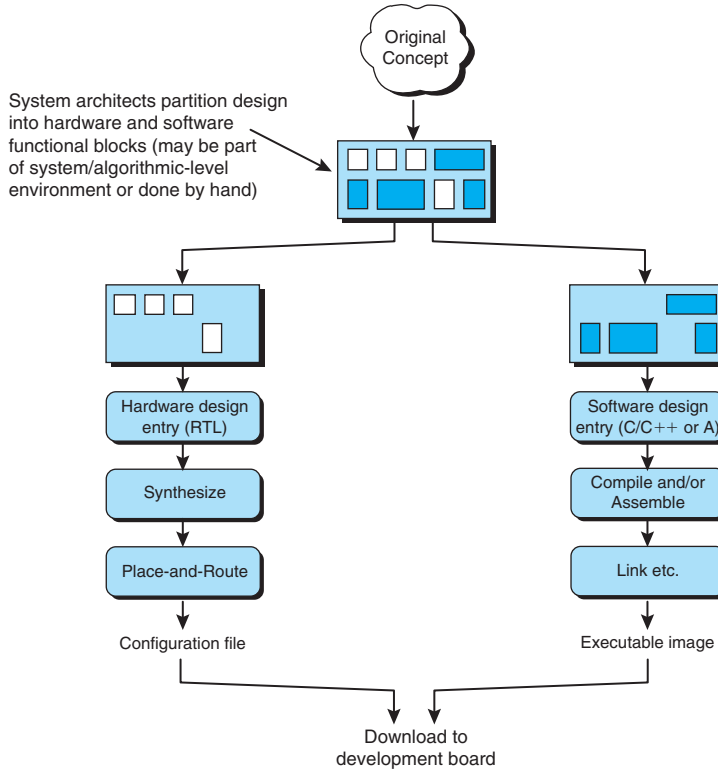

Original Concept
System architects partition design into hardware and software functional blocks (may be part of system/algorithmic-level environment or done by hand)
Hardware design entry (RTL)
Synthesize
Place-and-Route
Configuration file
Software design entry (C/C++ or A)
Compile and/or Assemble
Link etc.
Executable image
Download to development board

FIGURE 6-22 A (very) simple design flow.

One way to mitigate this is to use a virtual logic analyzer to observe what's happening in the hardware.

Things can be a little trickier when it comes to determining what's happening with the software. One point to remember is that—as discussed in Chapter 3—an embedded CPU core will have its own dedicated JTAG boundary scan chain (Figure 6-23).

This is true of both hard cores and the more sophisticated soft cores. In this case, the coverification environment can use the scan chain to monitor the activity on the buses and control signals connecting the CPU to the rest of the system. The CPU's internal registers can also be accessed via the JTAG port, thereby allowing an external debugger to take control of the device and single-step through instructions, set breakpoints, and so forth.

A Few Coverification Alternatives

If you really want to get visibility into what's happening in the hardware portions of design, one approach is to use a logic simulator. In this case, the

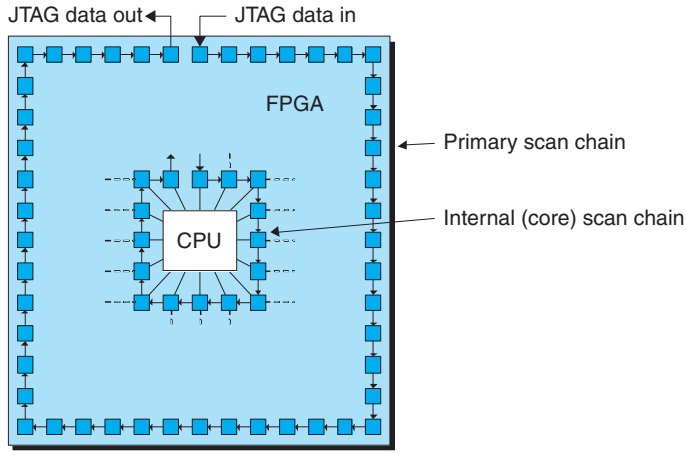


FIGURE 6-23 Embedded processor JTAG boundary scan chain.

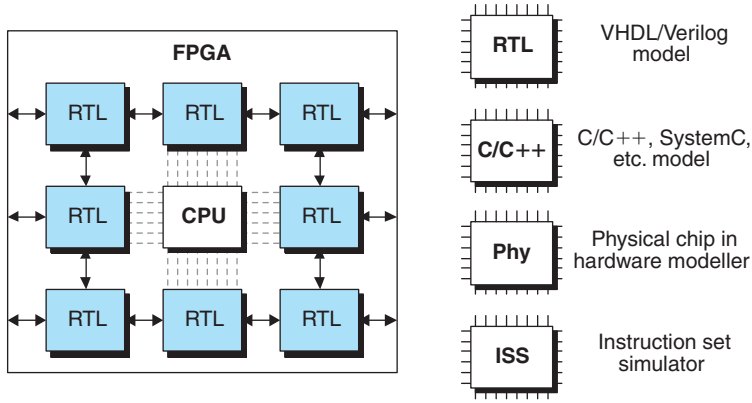


FIGURE 6-24 Alternative representations of the CPU.

majority of the system will be modeled and simulated in VHDL or Verilog/SystemVerilog at the RTL level of abstraction. When it comes to the CPU core, however, there are various ways in which to represent this (Figure 6-24).

Irrespective of the type of model used to represent the CPU, the embedded software (machine code) portion of the design will be loaded into some form of memory—either embedded memory in the FPGA or external memory devices—and the CPU model will then execute those machine code instructions.

Note that Figure 6-24 shows a high-level representation of the contents of the FPGA only. If the machine code is to be stored in external memory devices, then these devices would also have to be part of the simulation.

In fact, as a general rule of thumb, if the software talks to any stuff, then that stuff needs to be part of the coverification environment.

RTL (VHDL or Verilog)

Perhaps the simplest option here is when one has an RTL model of the CPU, in which case all of the activity takes place in the logic simulator. One disadvantage of this approach is that a CPU performs tremendous numbers of internal operations in order to perform the simplest task, which equates to incredibly slow simulation runs (you'll be lucky to be able to simulate 10 to 20 system clocks per second in real time).

The other disadvantage is that you have no visibility into what the software is doing at the source code level. All you'll be able to do is to observe logic values changing on wires and inside registers.

And there's always the fact that whoever supplies the real CPU doesn't want you to know how it works internally because that supplier may be using cunning proprietary tricks and wish to preserve their IP. In this case, you may well find it very difficult to lay your hands on an RTL model of the CPU at all.

C/C++, SystemC, etc.

As opposed to using an RTL model, it is very common to have access to some sort of C/C++ model of the CPU. (The proponents of SystemC have a vision of a world in which the CPU and the main peripheral devices all have SystemC models provided as standard for use in this type of design environment.)

The compiled version of this CPU model would be linked into the simulation via the programming language interface (PLI) in the case of a Verilog simulator or the foreign language interface (FLI)—or equivalent—in the case of a VHDL simulator.

The advantages of such a model are that it will run much faster than its RTL counterpart; that it can be delivered in compiled form, thereby preserving any secret IP; and that, at least in FPGA circles, such a model is usually provided for free (the FPGA vendors are trying to sell chips, not models).

One disadvantage of this approach is that the C/C++ model may not provide a 100-percent cycle-accurate representation of the CPU, which has the potential to cause problems if you aren't careful. But, once again, the main disadvantage of such a model is that its only purpose is to provide an engine to execute the machine code program, which means that you have no visibility into what the software is doing at the source code level. All you'll be able to do is observe logic values changing on wires and inside registers.

Physical Chip in Hardware Modeler

Yet another possibility is to use a physical device to represent a hard CPU core. For example, if you are using a PowerPC core in a Xilinx FPGA, you

can easily lay your hands on a real PowerPC chip. This chip can be installed in a box called a hardware modeler, which can then be linked into the logic simulation system.

The advantage of this approach is that you know the physical model (chip) is going to functionally match your hard core as closely as possible. Some disadvantages are that hardware modelers aren't cheap and they can be a pain to use.

The majority of hardware-modeler-based solutions don't support source-level debugging, which, once again, means that you have no visibility into what the software is doing at the source code level. All you'll be able to do is to observe logic values changing on wires and inside registers.

Instruction Set Simulator

As previously noted, in certain cases, the role of the software portion of a design may be somewhat limited. For example, the software may be acting as a state machine used to control some interface. Alternatively, the software's role may be to initialize certain aspects of the hardware and then sit back and watch the hardware do all of the work. If this is the case, then a C/C++ model or a physical model is probably sufficient—at least as far as the hardware design engineer is concerned.

At the other extreme, the hardware portions of the design may exist mainly to act as an interface with the outside world. For example, the hardware may read in a packet of data and store it in the FPGA's memory, and then the CPU may perform huge amounts of complex processing on this data. In cases like these, it is necessary for the software engineer to have sophisticated source-level debugging capabilities. This requires the use of an instruction set simulator (ISS), which provides a virtual representation of the CPU.

Although an ISS will almost certainly be created in C/C++, it will be architected very differently from the C/C++ models of the CPU discussed earlier in this section. This is because the ISS is created at a very high level of abstraction; it thinks in terms of transactions like "get me a word of data from location x in the memory," and it doesn't concern itself with details like how signals will behave in the real world.

How It Works

The easiest way to explain how this works is by means of an illustration (Figure 6-25).

First, the software engineers capture their program as C/C++ source code. This is then compiled using the -d (debug) option, which generates a symbol table and other debug-specific information along with the executable machine code image.

When we come to perform the coverification, there are a number of pieces to the puzzle. At one end we have the source-level debugger, whose interface is used by the software engineer to talk to the environment. At the other end we have the

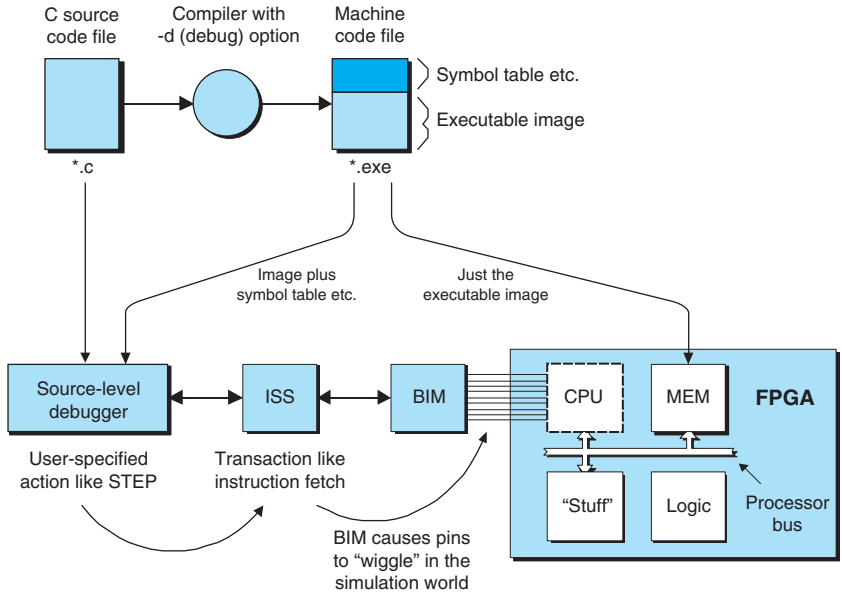


FIGURE 6-25 How an ISS fits into the picture.

logic simulator, which is simulating representations of the memory, stuff like peripheral devices, general-purpose logic, and so forth (for the sake of simplicity, this illustration assumes that all of the program memory resides in the FPGA itself).

In the case of the CPU, however, the logic simulator essentially sees a hole where this function should be. To be more precise, the simulator actually sees a set of inputs and outputs corresponding to the CPU. These inputs and outputs are connected to an entity called a bus interface model (BIM), which acts as a translator between the simulator and the ISS.

Both the source code and the executable image (along with the symbol table and other debug-centric information) are loaded into the source-level debugger. At the same time, the executable image is loaded into the MEM block. When the user requests the source-level debugger to perform an action like stepping through a line of source code, it issues commands to the ISS. In turn, the ISS will execute high-level transactions such as an instruction fetch, or a memory read/write, or an I/O command. These transactions are passed to the BIM, which causes the appropriate pins to "wiggle" in the simulation world.

Similarly, when something connected to the processor bus in the FPGA attempts to talk to the CPU, it will cause the pins driving the BIM to "wriggle." The BIM will translate these low-level actions into high-level transactions that it passes to the ISS, which will in turn inform the source-level debugger what's happening. The source-level debugger will then display the state of the program variables, the CPU registers, and other information of this ilk.

Insider Info

There are a variety of incredibly sophisticated (often frighteningly expensive) environments of this type on the market. Each has its own cunning tricks and capabilities, and some are more appropriate for ASIC designs than FPGAs or vice versa. As usual, however, this is a moving target, so you need to check around to see who is doing what before putting any of your precious money on the table.

INSTANT SUMMARY

The alternative FPGA design flows covered in this chapter were:

- SystemC-based flows
- Augmented C/C++-based flows
- Pure C/C++-based flows
- Mixed-language design/verification environments
- DSP-based flows using domain-specific languages
- DSP-based flows using system-level design/simulation environments
  - System/algorithmic level to RTL (both manual and automatic generation)
  - System/algorithmic level to C/C++
  - Block level IP environments
- Mixed DSP and VHDL/Verilog environments
- Embedded processor-based flows using hard cores
- Embedded processor-based flows using soft cores

Chapter 7

Using Design Tools

In an Instant

Simulation Tools
- Event-driven Logic Simulators
- Logic Values and Different Logic Value Systems
- Mixed-language Simulation
- Alternative Delay Formats
- Cycle-based Simulators
- Choosing a Logic Simulator

Synthesis (Logic/HDL versus Physically Aware)
- Logic/HDL Synthesis Technology
- Physically Aware Synthesis Technology
- Retiming, Replication, and Resynthesis

Timing Analysis
- Static Timing Analysis
- Statistical Static Timing Analysis

Verification in General
- Verification IP
- Verification Environments and Creating Testbenches
- Analyzing Simulation Results

Formal Verification
- Different Flavors of Formal Verification
- Terminology and Definitions
- Alternative Assertion/Property Specification Techniques
- Static Formal versus Dynamic Formal

Miscellaneous
- HDL to C Conversion
- Code Coverage
- Performance Analysis

Instant Summary

Definitions

Again we'll start with some basic design tool terms and definitions.

- *Event driven logic simulation* tools see the world as a series of discrete events.
- *Mixed language simulation* allows the use of multiple languages, such as Verilog and VHDL.
- *Logic synthesis* is a process in which a program is used to automatically convert a high-level textual representation of a design (specified using an HDL at the register transfer level (RTL) of abstraction) into equivalent registers and

Boolean equations. A synthesis tool automatically performs simplifications and minimizations and eventually outputs a gate-level netlist.

- *Physically aware synthesis* means taking actual placement information associated with the various logical elements in the design, using this information to estimate accurate track delays, and using these delays to fine-tune the placement and perform other optimizations.
- *Retiming* is a term used in the context of physical synthesis and is based on the concept of balancing out positive and negative slacks throughout the design, where *positive slack* refers to a path with some delay available that you're not using, and *negative slack* refers to a path that is using more delay than is available to it.
- *Replication* is similar to retiming, but focuses on breaking up long interconnect.
- *Resynthesis* uses the physical placement information to perform local optimizations on critical paths by means of operations like logic restructuring, reclustering, substitution, and possible elimination of gates and wires.
- *Formal verification* means using rigorous mathematical techniques and tools that employ such techniques to verify designs. In the not-so-distant past, this term was considered synonymous with *equivalency checking*.

SIMULATION TOOLS

Design engineers typically need to use a tremendous variety of tools to *capture*, *verify*, *synthesize*, and *implement* their designs. In this chapter we'll focus on some of the more significant contenders in the context of FPGA designs.

Event-driven Logic Simulators

Logic simulation is currently one of the main verification tools in the design (or verification) engineer's arsenal. The most common form of logic simulation is known as *event driven* because, perhaps not surprisingly, these tools see the world as a series of discrete events. As an example, consider a very simple circuit comprising an OR gate driving both a BUF (buffer) gate and a brace of NOT (inverting) gates, as shown in Figure 7-1.

Just to keep things simple, let's assume that NOT gates have a delay of 5 picoseconds (ps), BUF gates have a delay of 10 ps, and OR gates have a delay of 15 ps. On this basis, let's consider what will happen when a signal change occurs on one of the input pins (Figure 7-2).

Internally, the simulator maintains something called an *event wheel* onto which it places events that are to be "actioned" at some time in the future. When the first event occurs on input *in*1 at a time we might refer to as t_1, the simulator looks to see what this input is connected to, which happens to be

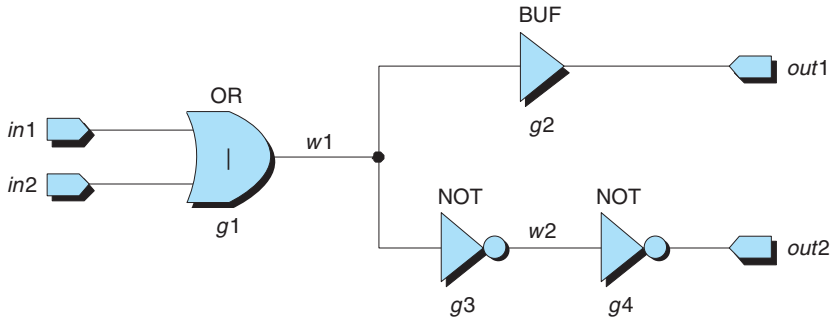


FIGURE 7-1 An example circuit.

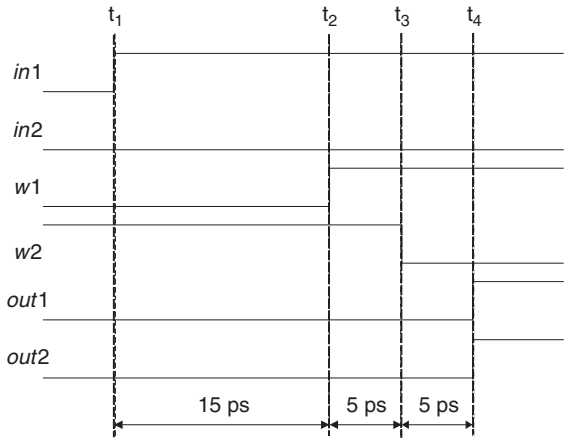


FIGURE 7-2 Results from an event-driven simulation.

our OR gate. We are assuming that the OR gate has a delay of 15 ps, so the simulator schedules an event on the output of the OR gate—a rising (0 to 1) transition on wire *w*1—for 15 ps in the future at time t_2.

The simulator then checks if any further actions need to be performed at the current time (t_1), then it looks at the event wheel to see what is to occur next. In the case of our example, the next event happens to be the one we just scheduled at time t_2, which was for a rising transition on wire *w*1. At the same time as the simulator is performing this action, it looks to see what wire *w*1 is connected to, which is BUF gate *g*2 and NOT gate *g*3.

As NOT gate *g*3 has a delay of 5 ps, the simulator schedules a falling (1 to 0) transition on its output, wire *w*2, for 5 ps in the future at time t_3. Similarly, as BUF gate *g*2 has a delay of 10 ps, the simulator schedules a rising (0 to 1) transition on its output, output *out*1, for 10 ps in the future at time t_4. And so it

goes until all of the events triggered by the initial transition on input *in*1 have been satisfied.

—Technology Trade-offs—

- The **advantage** of this event-driven approach is that simulators based on this technique can be used to represent almost any form of design, including synchronous and asynchronous circuits, combinatorial feedback loops, and so forth. These simulators also offer extremely good visibility into the design for debugging purposes, and they can evaluate the effects of delay-related narrow pulses and glitches that are very difficult to find using other techniques (see also the discussions on delays in the next section). The big **disadvantage** associated with these simulators is that they are extremely compute-intensive and correspondingly slow.

In the early days, event-driven digital logic simulators were simple tools that output results in the form of a textual (tabular) file. They evolved to a bit more advanced form, outputting results as graphical waveforms. Still later, the creators of digital simulators started to experiment with more sophisticated languages that could describe logical functions at higher levels of abstraction such as RTL. As the industry-standard HDLs such as Verilog and VHDL started to appear, they had the advantage that the same language could be used to represent both the functionality of the circuit and the testbench. (See also the discussions on special verification languages like *e* in the "Verification in General" section later in this chapter.)

Also, standard file formats for capturing simulation output results, such as the *value change dump* (VCD) format, started to appear on the scene. This facilitated third-party EDA companies creating sophisticated waveform display and analysis tools that could work with the outputs from multiple simulators. Similarly, innovations like the *standard delay format* (SDF) specification facilitated third-party EDA companies' creating sophisticated timing analysis tools that could evaluate circuits, generate timing reports highlighting potential problems, and output SDF files that could be used to provide more accurate timing simulations (see also the discussion on alternative delay formats below).

Logic Values and Different Logic Value Systems

The minimum set of logic values required to represent the operation of binary logic gates is 0 and 1. The next step is the ability to represent unknown values, for which we typically use the character *X*. These unknown values may be used to represent a variety of conditions, such as the contents of an uninitialized register or the clash resulting from two gates driving the same wire with opposing logical values. And it's also nice to be able to represent high-impedance values driven by the outputs of tri-state gates, for which we typically use the character *Z*.

ALERT!

As opposed to using the "*X*" character to represent "unknown" or "don't know," data books typically use it to represent "don't care." By comparison, hardware description languages tend to use "?" or "–" to represent "don't care" values. Also, "don't care" values cannot be assigned to outputs as driven states. Instead, they are used to specify how a model's inputs should respond to different combinations of signals.

But the 0, 1, *X*, and *Z* states are only the tip of the iceberg. More advanced logic simulators have ways to associate different drive strengths with the outputs of different gates. This is combined with ways in which to resolve and represent situations where multiple gates are driving the same wire with different logic values of different strengths. Just to make life fun, of course, VHDL and Verilog handle this sort of thing in somewhat different ways.

Mixed-language Simulation

The problem with having two industry-standard languages like Verilog and VHDL is that it's not long before you find yourself with different portions of a design represented in different languages. Anything you design from scratch will obviously be written in the language du jour favored by your company. However, problems can arise if you wish to reuse legacy code that is in the other language. Similarly, you may wish to purchase blocks of IP from a third party, but this IP may be available only in the language you aren't currently using yourself. And there's also the case where your company merges with, commences a joint project with, another company, where the two companies are entrenched in design flows using disparate languages.

There have historically been several flavors of *mixed-language simulation*, as described below:

- One technique used in the early days was to translate the "foreign" language (the one you weren't using) into the language you were working with. This was painful to say the least because the different languages supported different logic states and language constructs (even similar language statements had different semantics). The result was that when you simulated the translated design, it rarely behaved the way you expected it to, so this approach is rarely used today.
- Another technique was to have both a VHDL simulator and a Verilog simulator and to cosimulate the two simulation kernels. In this case the performance of the ensuing simulation was sadly lacking because each kernel was forever stopping while it waited for the other to complete an action. Thus, once again, this approach is rarely used today.

- The optimum solution is to have a single-kernel simulator that supports designs represented as a mixture of VHDL and Verilog blocks. All of the big boys in EDA have their own version of such a tool, and some go far beyond anything envisaged in the past because they can support multiple languages such as Verilog, SystemVerilog, VHDL, SystemC, and PSL (where PSL is introduced in more detail in the "Formal verification" section in this chapter).

Alternative Delay Formats

How you decide to represent delays in the models you are creating for use with an event-driven simulator depends on two things:

a. the delay modeling capabilities of the simulator itself and
b. where in the flow (and with what tools) you intend to perform your timing analysis.

A very common scenario is for *static timing analysis* (STA) to be performed externally from the simulation. In this case, logic gates (and more complex statements) may be modeled with zero (0 timebase unit) delays or unit (1 timebase unit) delays, where the term *timebase unit* refers to the smallest time segment recognized by the simulator.

Alternatively, we might associate more sophisticated delays with logic gates (and more complex statements) for use in the simulation itself. The first level of complexity is to separate rising delays from falling delays at the output from the gate (or more complex statement). For historical reasons, a rising (0-to-1) delay is often referred to as LH (standing for "low-to-high"). Correspondingly, a falling (1-to-0) delay may be referred to as HL (meaning "high-to-low"). For example, consider what happens if we were to apply a 12 ps positive-going (0-1-0) pulse to the input of a simple buffer gate with delays of LH = 5 ps and HL = 8 ps (Figure 7-3).

Not surprisingly, the output of the gate rises 5 ps after the rising edge is applied to the input, and it falls 8 ps after the falling edge is applied to the input. The really interesting point is that, due to the unbalanced delays, the 12 ps input pulse has been stretched to 15 ps at the output of the gate, where the additional 3 ps reflect the difference between the LH and HL values. Similarly, if a negative-going 12 ps (1-0-1) pulse were applied to the input of this gate, the corresponding pulse at the output would shrink to only 9 ps (try sketching this out on a piece of paper for yourself).

In addition to LH and HL delays, simulators also support minimum:typical:maximum (min:typ:max) values for each delay. For example, consider a positive-going pulse of 16 ps presented to the input of a buffer gate with rising and falling delays specified as 6:8:10 ps and 7:9:11 ps, respectively (Figure 7-4).

This range of values is intended to accommodate variations in the operating conditions such as temperature and voltage. It also covers variations in the

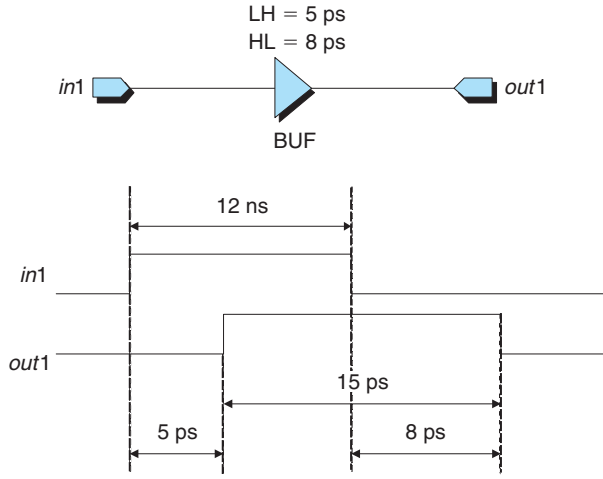


FIGURE 7-3 Separating LH and HL delays.

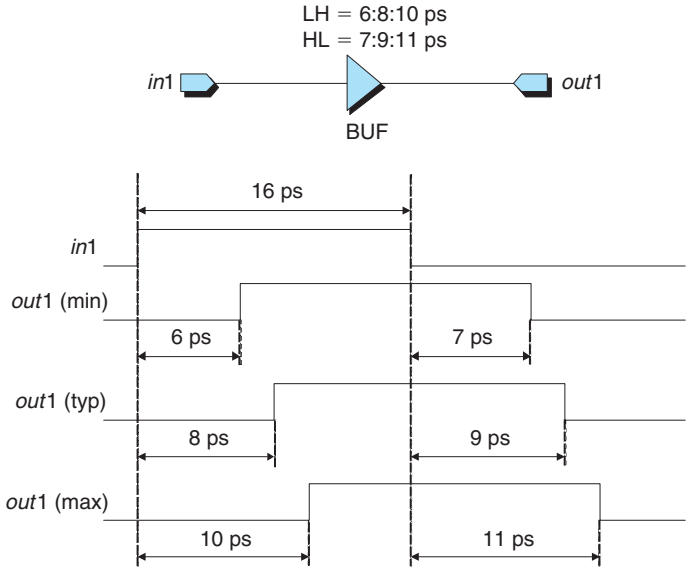


FIGURE 7-4 Supporting min:typ:max delays.

manufacturing process because some chips may run slightly faster or slower than others of the same type. Similarly, gates in one area of a chip (e.g., an ASIC or an FPGA) may switch faster or slower than identical gates in another area of the chip. (See also the discussions on timing analysis, particularly dynamic timing analysis, later in this chapter.)

Insider Info

In the early days, all of the input-to-output delays associated with a multi-input gate (or more complex statement) were identical. For example, consider a 3-input AND gate with an output called y and inputs a, b, and c. In this case, any LH and HL delays would be identical for the paths a-to-y, b-to-y, and c-to-y. Initially, this didn't cause any problems because it matched the way in which delays were specified in data books. Over time, however, data books began to specify individual input-to-output delays, so simulators had to be enhanced to support this capability.

Another point to consider is what will happen when a narrow pulse is applied to the input of a gate (or more complex statement). By "narrow" we mean a pulse that is smaller than the propagation delay of the gate. The first logic simulators were largely targeted toward simple ICs implemented in *transistor-transistor logic* (TTL) being used at the circuit board level. These chips typically rejected narrow pulses, so that's what the simulators did. This became known as the inertial delay model. As a simple example, consider two positive-going pulses of 8 ps and 4 ps applied to a buffer gate whose min:typ: max rising and falling delays are all set to 6 ps (Figure 7-5).

By comparison, logic gates implemented in later technologies such as *emitter-coupled logic* (ECL) would pass pulses that were narrower than the propagation delay of the gate. To accommodate this, some simulators were equipped with a

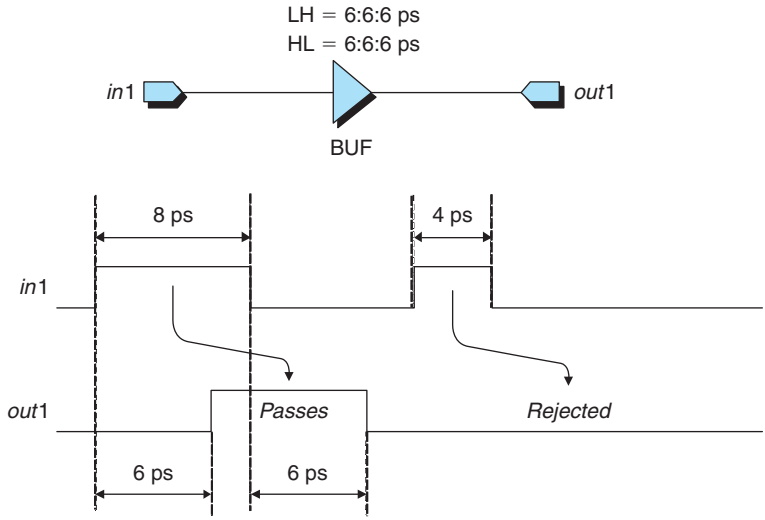


FIGURE 7-5 The inertial delay model rejects any pulse that is narrower than the gate's propagation delay.

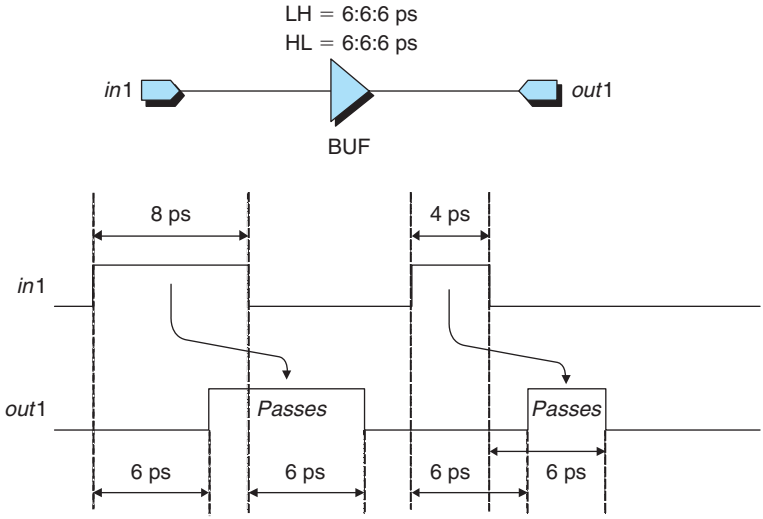


FIGURE 7-6 The transport delay model propagates any pulse, irrespective of its width.

mode called the transport delay model. Once again, consider two positive-going pulses of 8 ps and 4 ps applied to a buffer gate whose min:typ:max rising and falling delays are all set to 6 ps (Figure 7-6).

The problem with both the inertial and transport delay models is that they only provide for extreme cases, so the creators of some simulators started to experiment with more sophisticated narrow-pulse handling techniques, such as the *three-band delay model*. In this case, each delay may be qualified with two values called *r* (for "reject") and *p* (for "pass"), specified as percentages of the total delay. For example, assume we have a buffer gate whose min:typ:max delays have all been set to 6 ps qualified by *r* and *p* values of 33 percent and 66 percent, respectively (Figure 7-7).

Any pulses presented to the input that are greater than or equal to the *p* value will propagate; any pulses that are less than the *r* value will be completely rejected; and any pulses that fall between these two extremes will be propagated as a pulse with an unknown *X* value to indicate that they are ambiguous because we don't know whether they will propagate through the gate in the real world. (Setting both *r* and *p* to 100 percent equates to an inertial delay model, while setting them both to 0 percent reflects a pure transport delay model.)

Cycle-based Simulators

An alternative to the event-driven approach is to use a *cycle-based* simulation technique. This is particularly well suited to pipelined designs in which "islands" of combinational logic are sandwiched between blocks of registers (Figure 7-8).

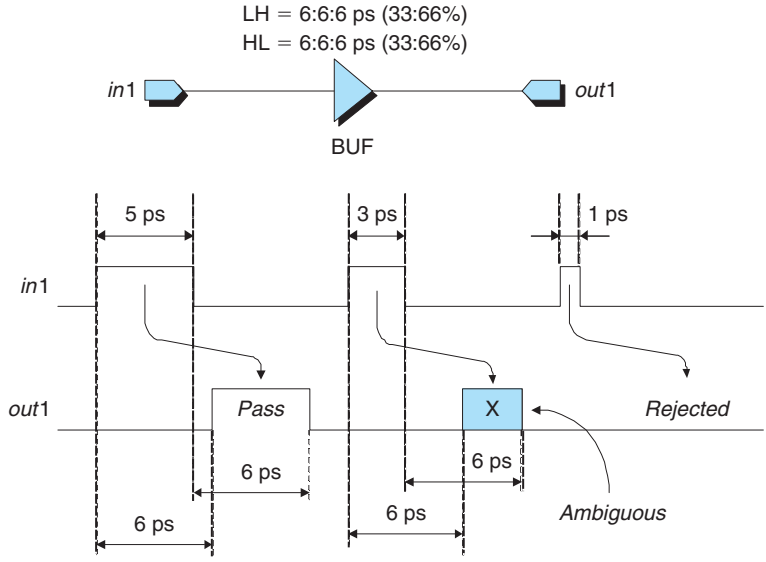


FIGURE 7-7 The three-band delay model.

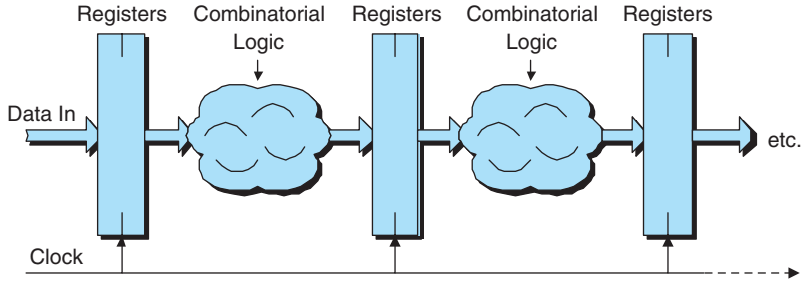


FIGURE 7-8 A simple pipelined design.

In this case, a cycle-based simulator will throw away any timing information associated with the gates forming the combinational logic and convert this logic into a series of Boolean operations that can be directly implemented using the CPU's logical machine code instructions.

—Technology Trade-offs—

- Given an appropriate circuit with appropriate activity, cycle-based simulators may offer significant **run-time advantages** over their event-driven counterparts. The **downside**, however, is that they typically only work with 0 and 1 logic values (no *X* or *Z* values, and no drive strength representations). Also, cycle-based simulators can't represent asynchronous logic or combinatorial feedback loops.

These days it's rare to see anyone using a pure cycle-based simulator. However, several event-driven simulators have been augmented to have hybrid capabilities. In this case, if you instruct the simulator to aim for extreme performance (as opposed to timing accuracy), it will automatically handle some portions of the circuit using an event-driven approach and other portions using cycle-based techniques.

Choosing a Logic Simulator

Choosing a logic simulator is, as with anything else in engineering, a balancing act. Here are some things to consider:

1. Think about whether you require **mixed-language capability**. If you are a small startup, you may be planning to use only one language, but remember that any IP you decide to purchase down the road may not be available in this language. Having a solution that can work with VHDL, Verilog, and SystemVerilog would be a good start, and if it can also handle SystemC along with one or more formal verification languages, then it will probably stand you in good stead for some time to come.
2. Generally speaking, **performance** is the number-one criterion for most folks. The trick here is how to determine the performance of a simulator without being bamboozled. The only way to really do this is to have your own benchmark design and to run it on a number of simulators. Creating a good benchmark design is a nontrivial exercise, but it's much better than using a design supplied by an EDA vendor (because such a design will be tuned to favor their solution, while delivering a swift knee to the metaphorical groins of competing tools).
3. However, there's more to life than raw performance. You also need to look for a good **interactive debugging solution** such that when you detect a problem, you can stop the simulator and poke around the design. All simulators are not created equal in this department. In some cases, even if the simulator does let you do what you want, you may have to jump through hoops to get there. So the trick here is—after running your performance benchmark—bring up the same circuit with a known bug and see how easy it is (and how long it takes) to detect and isolate the little rapscallion. In reality, some simulators that give you the performance you require do such a poor job in this department that you are obliged to use third-party postsimulation analysis tools.
4. Another thing to consider is the **capacity of the simulator**. The tools supplied by the big boys in EDA essentially have no capacity limitations, but simulators from smaller vendors might be based on ported 32-bit code if you were to look under the hood. Of course, if you are only going to work with smaller designs (say, equivalent to 500,000 gates or less), then you will probably be okay with the simulators supplied by the FPGA

vendors (these are typically "lite" versions of the tools supplied by the big EDA vendors).

Of course, you will have your own criteria in addition to the topics raised above, such as the **quality of the code coverage** and **performance analysis** provided by the various tools. These used to be the province of specialist third-party tools, but most of the larger simulators now provide some level of integrated code coverage and performance analysis in the simulation environment itself. However, different simulators offer different feature sets (see also the discussions on code coverage and performance analysis in the "Miscellaneous" section later in this chapter).

SYNTHESIS (LOGIC/HDL VERSUS PHYSICALLY AWARE)

Logic/HDL Synthesis Technology

Traditional *logic synthesis* tools appeared on the scene around the early to mid-1980s. Depending on whom you are talking to, these tools are now often referred to as HDL synthesis technology.

The role of the original logic/HDL synthesis tools was to take an RTL representation of an ASIC design along with a set of timing constraints and to generate a corresponding gate-level netlist. During this process, the synthesis application performed a variety of minimizations and optimizations (including optimizing for area and timing).

Around the middle of the 1990s, synthesis tools were augmented to understand the concept of FPGA architectures. These architecturally aware applications could output a LUT/CLB-level netlist, which would subsequently be passed to the FPGA vendor's place-and-route software (Figure 7-9).

—Technology Trade-offs—

- In real terms, the FPGA designs generated by architecturally aware synthesis tools were 15 to 20 percent faster than their counterparts created using traditional gate-level synthesis offerings.

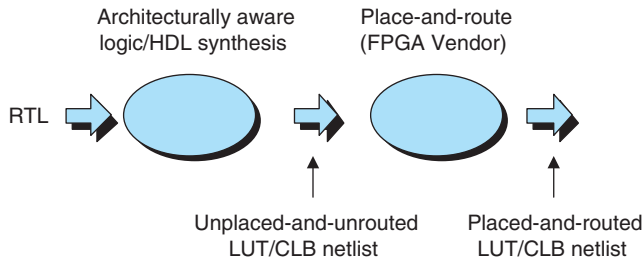


FIGURE 7-9 Traditional logic/HDL synthesis.

Physically Aware Synthesis Technology

The problem with traditional logic/HDL synthesis is that it was developed when logic gates accounted for most of the delays in a timing path, while track delays were relatively insignificant. This meant that the synthesis tools could use simple wire-load models to evaluate the effects of the track delays. (These models were along the lines of: One load gate on a wire equates to *x* pF of capacitance; two load gates on a wire equates to *y* pF of capacitance; etc.) The synthesis tool would then estimate the delay associated with each track as a function of its load and the strength of the gate driving the wire.

This technique was adequate for the designs of the time, which were implemented in multimicron technologies and which contained relatively few logic gates by today's standards. By comparison, modern designs can contain tens of millions of logic gates, and their deep submicron feature sizes mean that track delays can account for up to 80 percent of a delay path. When using traditional logic/HDL synthesis technology on this class of design, the timing estimations made by the synthesis tool bear so little resemblance to reality that achieving timing closure can be well-nigh impossible.

For this reason, ASIC flows started to see the use of *physically aware synthesis* somewhere around 1996, and FPGA flows began to adopt similar techniques circa 2000 or 2001.

FAQ

What does "physically aware" really mean?

Of course there is a variety of different definitions as to exactly what the term physically aware synthesis implies. The core concept is to use physical information earlier in the synthesis process, but what does this actually mean? For example, some companies have added interactive floor-planning capabilities to the front of their synthesis engines, and they class this as being physical synthesis or physically aware synthesis. For most folks, however, physically aware synthesis means taking actual placement information associated with the various logical elements in the design, using this information to estimate accurate track delays, and using these delays to fine-tune the placement and perform other optimizations. Interestingly enough, physically aware synthesis commences with a firstpass run using a relatively traditional logic/HDL synthesis engine (Figure 7-10).

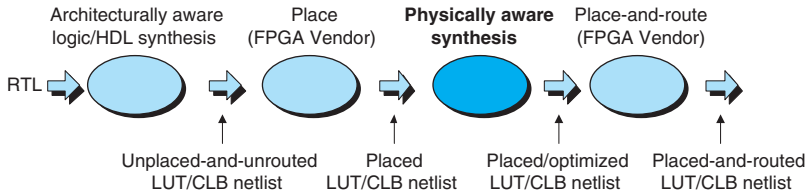


FIGURE 7-10 Physically aware synthesis.

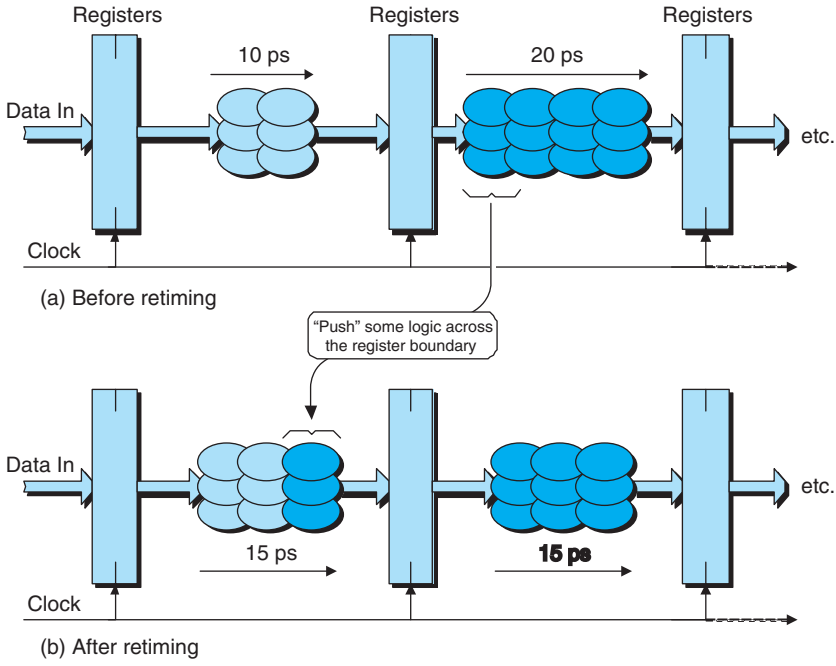


FIGURE 7-11 Retiming.

Retiming, Replication, and Resynthesis

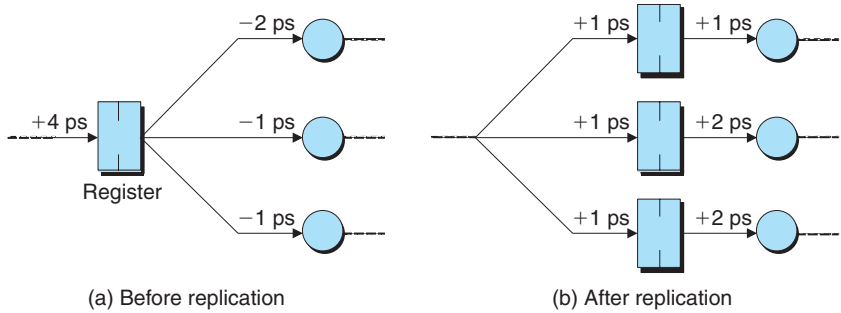
In this section, we'll discuss several concepts related to physical synthesis that were defined earlier: *retiming*, *replication*, and *resynthesis*.

As an example, let's assume a pipelined design whose clock frequency is such that the maximum register-to-register delay is 15 ps. Now let's assume that we have a situation as shown in Figure 7-11a, whereby the longest timing path in the first block of combinational logic is 10 ps (which means it has a *positive slack* of 5 ps), while the longest path in the next block of combinational logic is 20 ps (which means it has a *negative slack* of 5 ps).

Once the initial path timing, including routing delays, has been calculated, combinational logic is moved across register boundaries (or vice versa, depending on your point of view) to steal from paths with positive slack and donate to paths with negative slack (Figure 7-11b). Retiming is very common in physically aware FPGA design flows because registers are plentiful in FPGA devices.

Replication is similar to retiming, but it focuses on breaking up long interconnect. For example, let's assume that we have a register with 4 ps of positive slack on its input. Now let's assume that this register is driving three paths, whose loads each see negative slack (Figure 7-12a).

By replicating the register and placing the copies close to each load, we can redistribute the slack to make all of the timing paths work (Figure 7-12b).



(a) Before replication (b) After replication

FIGURE 7-12 Replication.

Last, but not least, the concept of *resynthesis* is based on the fact that there are many different ways of implementing (and placing) different functions. Resynthesis uses the physical placement information to perform local optimizations on critical paths by means of operations like logic restructuring, reclustering, substitution, and possible elimination of gates and wires.

Insider Info

In the real world, the capabilities of the various synthesis engines, along with associated features like autointeractive floor planning, change on an almost daily basis, and the various vendors are constantly leapfrogging each other. There's also the fact that different engines may work better (or worse) with different FPGA vendors' architectures. One thing to look for is the ability (or lack thereof) of the engine to infer things automatically, like clocking elements and embedded functions, from your source code or constraints files without your having to define them explicitly.

TIMING ANALYSIS

Static Timing Analysis

The most common form of timing verification in use today is classed as STA. Conceptually, this is quite simple, although in practice things are, as usual, more complex than they might at first appear.

The timing analyzer essentially sums all of the gate and track delays forming each path to give you the total input-to-output delays for each path. (In the case of pipelined designs, the analyzer calculates delays from one bank of registers to the next.)

Prior to place-and-route, the analyzer may make estimations as to track delays. Following place-and-route, the analyzer will employ extracted parasitic values (for resistance and capacitance) associated with the physical tracks to provide more accurate results. The analyzer will report any paths that fail to

meet their original timing constraints, and it will also warn of potential timing problems (e.g., setup and hold violations) associated with signals being presented to the inputs of any registers or latches.

—Technology Trade-offs—

- STA is particularly well suited to classical synchronous designs and pipelined architectures. The main **advantages** of STA are that it is relatively fast, it doesn't require a test bench, and it exhaustively tests every possible path into the ground. **On the downside**, static timing analyzers are little rascals when it comes to detecting false paths that will never be exercised during the course of the design's normal operation. Also, these tools aren't at their best with designs employing latches, asynchronous circuits, and combinational feedback loops.

Statistical Static Timing Analysis

STA is a mainstay of modern ASIC and FPGA design flows, but it's starting to run into problems with the latest process technology nodes. At the time of writing, an increasing number of folks are starting to design at the 45-nano node, with the 32-nano node expected to see mainstream adoption starting around 2011/2012.

As previously discussed, in the case of modern silicon chips, interconnect delays dominate logic delays, especially with respect to FPGA architectures. In turn, interconnect delays are dependent on parasitic capacitance, resistance, and inductance values, which are themselves functions of the topology and cross-sectional shape of the wires.

The problem is that, in the case of the latest technology process nodes, photolithographic processes are no longer capable of producing exact shapes. Thus, as opposed to working with squares and rectangles, we are now working with circles and ellipsoids. Feature sizes like the widths of tracks are now so small that small variations in the etching process cause deviations that, although slight, are significant with relation to the main feature size. (These irregularities are made more significant by the fact that in the case of high-frequency designs, the so-called skin-effect comes into play, which refers to the fact that high-frequency signals travel only through the outer surface, or skin, of the conductor.) Furthermore, there are variations in the vertical plane of the track's cross section caused by processes like *chemical mechanical polishing* (CMP).

As an overall result, it's becoming increasingly difficult to calculate track delays accurately. Of course, it is possible to use the traditional engineering fallback of guard-banding (using worst-case estimations), but excessively conservative design practices result in device performance significantly below the silicon's full potential, which is an extremely unattractive option in today's highly competitive marketplace. In fact, the effects of geometry variations are causing the probability distributions of delays to become so wide that worst-case numbers may actually be slower than in an earlier process technology!

One potential solution is the concept of the *statistical static timing analyzer* (SSTA). This is based on generating a probability function for the delay associated with each signal for each segment of a track, then evaluating the total delay probability functions of signals as they propagate through entire paths. The problem is that SSTA is very complex and the distribution functions are—in reality—not nice Gaussian curves. Just to add to the fun and frivolity, some of the distribution functions tend to be time-based; as a piece of equipment at the foundry undergoes "wear" over time, for example, this can affect some of the probability distributions. Having said this, by 2008 most of the "big boys" supplying tools to design and verify integrated circuits (e.g., Cadence, Magma, Synopsys, etc.) had an SSTA offering of one form or another (some are better than others). Many of the folks designing chips at the 45-nano node are using SSTA, and many observers believe that the use of SSTA will be mandatory at the forthcoming 32-nano node (actually, in addition to timing analysis, statistical techniques are starting to appear in other analysis engines, such as power consumption and noise/signal integrity).

VERIFICATION IN GENERAL

As designs increase in complexity, verifying their functionality consumes more and more time and resources. Such verification includes implementing a verification environment, creating a testbench, performing logic simulations, analyzing the results to detect and isolate problems, and so forth. In fact, verifying one of today's high-end ASIC, SoC, or FPGA designs can consume 70 percent or more of the total development effort from initial concept to final implementation.

Verification IP

One way to alleviate this problem is to make use of *verification IP*. The idea here is that the design, which is referred to as the *device under test* (DUT) for the purposes of verification, typically communicates with the outside world using standard interfaces and protocols. Furthermore, the DUT is typically communicating with devices such as microprocessors, peripherals, arbiters, and the like.

The most commonly used technique for performing functional verification is to use an industry-standard event-driven logic simulator. One way to test the DUT would be to create a testbench describing the precise bit-level signals to be applied to the input pins and the bit-level responses expected at the outputs. However, the protocols for the various interfaces and buses are now so complex that it is simply not possible to create a test suite in this manner.

Another technique would be to use RTL models of all of the external devices forming the rest of the system. However, many of these devices are extremely proprietary and RTL models may not be readily available. Furthermore, simulating an entire system using fully functional models of all

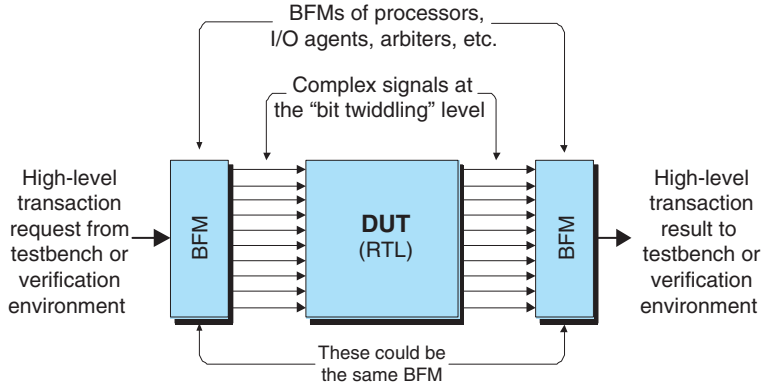


FIGURE 7-13 Using verification IP in the form of BFMs.

of the processor and I/O devices would be prohibitively expensive in terms of time and computing requirements.

The solution is to use verification IP in the form of *bus functional models* (BFMs) to represent the processors and the I/O agents forming the system under test (Figure 7-13).

A BFM doesn't replicate the entire functionality of the device it represents; instead, it emulates the way the device works at the bus interface level by generating and accepting transactions. In this context, the term transaction refers to a high-level bus event such as performing a read or write cycle. The verification environment (or testbench) can instruct a BFM to perform a specific transaction like a memory write. The BFM then generates the complex low-level ("bit-twiddling") signal interactions on the bus driving the DUT's interface transparently to the user.

Similarly, when the DUT (the design) responds with a complex pattern of signals, another BFM (or maybe the original BFM) can interpret these signals and translate them back into corresponding high-level transactions. (See also the discussions on verification environments and creating testbenches below.)

Key Concept

It should be noted that, although they are much smaller and simpler (and hence simulate much faster) than fully functional models of the devices they represent, BFMs are by no means trivial. For example, sophisticated BFMs, which are often created as cycle-accurate, bit-accurate C/C++ models, may include internal caches (along with the ability to initialize them), internal buffers, configuration registers, write-back queues, and so forth. Also, BFMs can provide a tremendous range of parameters that provide low-level control of such things as address timing, snoop timing, data wait states for different memory devices, and the like.

Verification Environments and Creating Testbenches

When I was a young man starting out in simulation, we created test vectors (stimulus and response) to be used with our simulations as tabular ASCII text files containing logic 0 and 1 values (or hexadecimal values if you were lucky). At that time, the designs we were trying to test were incredibly simple compared to today's monsters, so an English translation of our tests would be something along the lines of:

At time 1,000 make the reset signal go into its active state.
At time 2,000 make the reset signal go into its inactive state.
At time 2,500 check to see that the 8-bit data bus is 00000000.
At time … and so it went.

Over time, designs became more complex, and the way in which they could be verified became more sophisticated with the advent of high-level languages that could be used to specify stimulus and expected response. These languages sported a variety of features such as loop constructs and the ability to vary the tests depending on the state of the outputs (e.g., "If the status bus has a value of 010, then jump to test xyz"). At some stage, folks started referring to these tests as *testbenches*.

Insider Info

To be a tad more pedantic, the term "testbench" really refers to the infrastructure supporting test execution.

The current state of play is that many of today's designs are now so complex that it's well nigh impossible to create an adequate testbench by hand. This has paved the way for sophisticated verification environments and languages. Perhaps the most sophisticated of the languages, known by some as *hardware verification languages* (HVLs), is the aspect-oriented ***e*** offering from Verisity Design (www.verisity.com).

In case you were wondering, ***e*** doesn't actually stand for anything now, but originally it was intended to reflect the idea of "English-like" in that it has a natural language feel to it. You can use ***e*** to specify directed tests if you wish, but you would typically only wish to do this for special cases. Instead, the concept behind ***e***, which you can think of as a blend of C and Verilog with a hint of Pascal, is more about declaring valid ranges and sequences of input values (along with their invalid counterparts) and high-level verification strategies. This ***e*** description is then used by an appropriate verification environment to guide the simulations.

Analyzing Simulation Results

Almost every simulator comes equipped with a graphical waveform viewer that can be used to display results interactively (as the simulator runs) or to accept and display postsimulation results from a *value change dump* (VCD) file.

Sad to relate, however, some of these tools are not as effective as one might hope when it comes to really analyzing this information and tracking down problems. In this case, you might wish to use a tool from a third-party vendor.

FORMAL VERIFICATION

Although large computer and chip companies like IBM, Intel, and Motorola have been developing and using formal tools internally for decades (since around the mid-1980s), the whole field of *formal verification* (FV) is still relatively new to a lot of folks. This is particularly true in the FPGA arena, where the adoption of formal verification is lagging behind its use in ASIC design flows. Having said this, formal verification can be such an incredibly powerful tool that more and more folks are starting to use it in earnest.

One big problem is that formal verification is still so new to mainstream usage that there are many players, all of whom are happily charging around in a bewildering variety of different directions. Also, as opposed to a lack of standards, there are now so many different offerings that the mind boggles. The confusion is only increased by the fact that almost everyone you talk to puts his or her unique spin on things (if, for example, you ask 20 EDA vendors to define and differentiate the terms assertion and property, your brains will leak out of your ears at the diametrically opposing responses).

Trying to unravel this morass is a daunting task to say the least. However, there is nothing to fear but fear itself, as my dear old dad used to say, so let's take a stab at rending the veils asunder and describing formal verification in a way that we can all understand.

Different Flavors of Formal Verification

As mentioned at the beginning of this chapter, the term *formal verification* was considered synonymous with equivalency checking for the majority of design engineers. In this context, an equivalency checker is a tool that uses formal (rigorous mathematical) techniques to compare two different representations of a design—say an RTL description with a gate-level netlist—to determine whether they have the same input-to-output functionality.

In fact, equivalency checking may be considered a subclass of formal verification called *model checking*, which refers to techniques used to explore the state-space of a system to test whether certain properties, typically specified in

the form of assertions, are true. (Definitions of terms like property and assertion are presented a little later in this section.)

For the purposes of the remainder of our discussions here, we shall understand formal verification to refer to model checking. It should be noted, however, that there is another category of formal verification known as *automated reasoning*, which uses logic to prove, much like a formal mathematical proof, that an implementation meets an associated specification.

FAQs

What is formal verification, and why is it so cool?

To provide a starting point for our discussions, let's assume we have a design comprising a number of subblocks and that we are currently working with one of these blocks, whose role in life is to perform some specific function. In addition to the HDL representation that defines the functionality of this block, we can also associate one or more assertions/properties with that block (these assertions/properties may be associated with signals at the interface to the block or with signals and registers internal to the block).

A very simple assertion/property might be along the lines of "Signals A and B should never be active (low) at the same time." But these statements can also extend to extremely complex transaction-level constructs, such as "When a PCI write command is received, then a memory write command of type xxxx must be issued within 5 to 36 clock cycles."

Thus, assertions/properties allow you to describe the behavior of a time-based system in a formal and rigorous manner that provides an unambiguous and universal representation of the design's intent. Furthermore, assertions/properties can be used to describe both expected and prohibited behavior.

The fact that assertions/properties are both human and machine-readable makes them ideal for the purposes of capturing an executable specification, but they go far beyond this. Let's return to considering a very simple assertion/property such as "Signals A and B should never be active (low) at the same time." One term you will hear a lot is *assertion-based verification* (ABV), which comes in several flavors: *simulation, static formal verification,* and *dynamic formal verification*.

In the case of static formal verification, an appropriate tool reads in the functional description of the design (typically at the RTL level of abstraction) and then exhaustively analyzes the logic to ensure that this particular condition can never occur.

By comparison, in the case of dynamic formal verification, an appropriately augmented logic simulator will sum up to a certain point, then pause and automatically invoke an associated formal verification tool (this is discussed in more detail below).

Of course, assertions/properties can be associated with the design at any level, from individual blocks, to the interfaces linking blocks, to the entire system. This leads to a very important point, that of *verification reuse*.

Key Concept

Prior to formal verification, there was very little in the way of verification reuse. For example, when you purchase an IP core, it will typically come equipped with an associated testbench that focuses on the I/O signals at the core's boundary. This allows you to verify the core in isolation, but once you've integrated the core into the middle of your design, its testbench is essentially useless to you. Now consider purchasing an IP core that comes equipped with a suite of predefined assertions/properties, like "Signal A should never exhibit a rising transition within three clocks of Signal B going active." These assertions/properties provide an excellent mechanism for communicating interface assumptions from the IP developer to downstream users. Furthermore, these assertions/properties remain true and can be evaluated by the verification environment, even when this IP core is integrated into your design.

With regard to assertions/properties associated with the system's primary inputs and outputs, the verification environment may use these to automatically create stimuli to drive the design. Furthermore, you can use assertions/properties throughout the design to augment code and functional coverage analysis (see also the "Miscellaneous" section below) to ensure that specific sequences of actions or conditions have been performed.

Terminology and Definitions

Now that we've discussed the overall concept of the model checking aspects of formal verification, we are better equipped to wade through some further terminology and definitions. To be fair, this is relatively uncharted water; the following was gleaned from talking with lots of folks and then desperately trying to rationalize the discrepancies between the tales they told.

- **Assertions/properties**: The term property comes from the model checking domain and refers to a specific functional behavior of the design that you want to (formally) verify (e.g., "after a request, we expect a grant within 10 clock cycles"). By comparison, the term assertion stems from the simulation domain and refers to a specific functional behavior of the design that you want to monitor during simulation (and flag a violation if that assertion "fires").

  Today, with the use of formal tools and simulation tools in unified environments and methodologies, the terms property and assertion tend to be used interchangeably; that is, a property is an assertion and vice versa. In general, we understand an assertion/property to be a statement about a specific attribute associated with the design that is expected to be true. Thus, assertions/properties can be used as checkers/monitors or as targets of formal proofs, and they are usually used to identify/trap undesirable behavior.

- **Constraints**: The term constraint also derives from the model checking space. Formal model checkers consider all possible allowed input combinations when performing their magic and working on a proof. Thus, there is often a need to constrain the inputs to their legal behavior; otherwise, the tool would report false negatives, which are property violations that would not normally occur in the actual design.

  As with properties, constraints can be simple or complex. In some cases, constraints can be interpreted as properties to be proven. For example, an input constraint associated with one module could also be an output property of the module driving this input. So, properties and constraints may be dual in nature. (The term constraint is also used in the "constrained random simulation" domain, in which case the constraint is typically used to specify a range of values that can be used to drive a bus.)
- **Event**: An event is similar to an assertion/property, and in general events may be considered a subset of assertions/properties. However, while assertions/properties are typically used to trap undesirable behavior, events may be used to specify desirable behavior for the purposes of functional coverage analysis.

  In some cases, assertions/properties may consist of a sequence of events. Also, events can be used to specify the window within which an assertion/property is to be tested (e.g., "After *a*, *b*, *c*, we expect *d* to be true, until *e* occurs," where *a*, *b*, *c*, and *e* are all events, and *d* is the behavior being verified). Measuring the occurrence of events and assertions/properties yields quantitative data as to which corner cases and other attributes of the design have been verified. Statistics about events and assertions/properties can also be used to generate functional coverage metrics for a design.
- **Procedural**: The term procedural refers to an assertion/property/event/constraint that is described within the context of an executing process or set of sequential statements, such as a VHDL process or a Verilog "always" block (thus, these are sometimes called "incontext" assertions/properties). In this case, the assertion/property is built into the logic of the design and will be evaluated based on the path taken through a set of sequential statements.
- **Declarative**: The term declarative refers to an assertion/property/event/constraint that exists within the structural context of the design and is evaluated along with all of the other structural elements in the design (for example, a module that takes the form of a structural instantiation). Another way to view this is that a declarative assertion/property is always "on/active," unlike its procedural counterpart that is only "on/active" when a specific path is taken/executed through the HDL code.
- **Pragma**: The term pragma is an abbreviation for "pragmatic information," which refers to special pseudocomment directives that can be interpreted and used by parsers/compilers and other tools. (Note that this is a general-purpose term, and pragma-based techniques are used in a variety of tools in addition to formal verification technology.)

Alternative Assertion/Property Specification Techniques

This is where the fun really starts, because there are various ways in which assertions/properties and so forth can be implemented, as summarized below:

- ***Special languages***: This refers to using a formal property/assertion language that has been specially constructed for the purpose of specifying assertions/ properties with maximum efficiency. Languages of this type, of which Sugar, PSL, and OVA are good examples, are very powerful in creating sophisticated, regular, and temporal expressions, and they allow complex behavior to be specified with very little code (Sugar, PSL, and OVA are introduced in more detail later in this chapter).

  Such languages are often used to define assertions/properties in "side-files" that are maintained outside the main HDL design representation. These side-files may be accessed during parser/compile time and implemented in a declarative fashion. Alternatively, a parser/compiler/simulator may be augmented to allow statements in the special language to be embedded directly in the HDL as in-line code or as pragmas; in both of these cases, the statements may be implemented in a declarative and/or procedural manner.
- ***Special statements in the HDL itself***: Right from the get-go, VHDL came equipped with a simple assert statement that checks the value of a Boolean expression and displays a user-specified text string if the expression evaluates False. The original Verilog did not include such a statement, but SystemVerilog has been augmented to include this capability.

  The advantage of this technique is that these statements are ignored by synthesis engines, so you don't have to do anything special to prevent them from being physically implemented as logic gates in the final design. The disadvantage is that they are relatively simplistic compared to special assertion/property languages and are not well equipped to specify complex temporal sequences (although SystemVerilog is somewhat better than VHDL in this respect).
- ***Models written in the HDL and called from within the HDL***: This concept refers to having access to a library of internally or externally developed models. These models represent assertions/properties using standard HDL statements, and they may be instantiated in the design like any other blocks. However, these instantiations will be wrapped by synthesis OFF/ ON pragmas to ensure that they aren't physically implemented. A good example of this approach is the open verification library (OVL) from the Accellera standards committee (www.accellera.org), as discussed in the next section.
- ***Models written in the HDL and accessed via pragmas***: This is similar in concept to the previous approach in that it involves a library of models that represent assertions/properties using standard HDL statements. However, as opposed to instantiating these models directly from the main design code,

they are pointed to by pragmas. A good example of this technique is the CheckerWare® library from 0-In Design Automation (www.0-In.com). For example, consider a design containing the following line of Verilog code:

```
reg [5:0] STATE_VAR; // 0in one_hot
```

The left-hand side of this statement declares a 6-bit register called STATE_VAR, which we can assume is going to be used to hold the state variables associated with an FSM. Meanwhile, the right-hand side ("0in one_hot") is a pragma. Most tools will simply treat this pragma as a comment and ignore it, but 0-In's tools will use it to call a corresponding "one-hot" assertion/property model from their CheckerWare library. Note that the 0-In implementation means that you don't need to specify the variable, the clocking, or the bit-width of the assertion; this type of information is all picked up automatically. Also, depending on a pragma's position in the code, it may be implemented in a declarative or procedural manner.

Static Formal versus Dynamic Formal

This is a little tricky to wrap one's brain around, so let's take things step by step. First, you can use assertions/properties in a simulation environment. In this case, if you have an assertion/property along the lines of "Signals A and B should never be active (low) at the same time," then if this illegal case occurs during the course of a simulation, a warning flag will be raised, and the fact this happened can be logged.

Simulators can cover a lot of ground, but they require some sort of test-bench or a verification environment that is dynamically generating stimulus. Another consideration is that some portions of a design are going to be difficult to verify via simulation because they are deeply buried in the design, making them difficult to control from the primary inputs. Alternatively, some areas of a design that have large amounts of complex interactions with other state machines or external agents will be difficult to control.

At the other end of the spectrum is *static formal verification*. These tools are incredibly rigorous and they examine 100 percent of the state space without having to simulate anything. Their disadvantage is that they can typically be used for small portions of the design only, because the state space increases exponentially with complex properties, and one can quickly run into a "state space explosion." By comparison, logic simulators, which can also be used to test for assertions, can cover a lot of ground, but they do require stimuli, and they don't cover every possible case.

To address these issues, some solutions combine both techniques. For example, they may use simulation to reach a corner condition and then automatically pause the simulator and invoke a static formal verification engine to

exhaustively evaluate that corner condition. (In this context, a general definition of a "corner condition" or "corner case" is a hard-to-exercise or hard-to-reach functional condition associated with the design.) Once the corner condition has been evaluated, control will automatically be returned to the simulator, which will then proceed on its merry way. This combination of simulation and traditional static formal verification is referred to as dynamic formal verification.

As one simple example of where this might be applicable, consider a FIFO memory, whose "Full" and "Empty" states may be regarded as corner cases. Reaching the "Full" state will require many clock cycles, which is best achieved using simulation. But exhaustively evaluating attributes/properties associated with this corner case, such as the fact that it should not be possible to write any more data while the FIFO is full, is best achieved using static techniques.

Once again, a good example of this dynamic formal verification approach is provided by 0-In. Corner cases are explicitly defined as such in their CheckerWare library models. When a corner case is reached during simulation, the simulator is paused, and a static tool is used to analyze that corner case in more detail.

FAQ

Is there a standard formal verification language?

Let's begin with something called Vera®, which began life with work done at Sun Microsystems in the early 1990s. It was provided to Systems Science Corporation somewhere around the mid-1990s, which was in turn acquired by Synopsys in 1998. Vera is essentially an entire verification environment, similar to, but perhaps not as sophisticated as, the e verification language/environment introduced earlier in this chapter. Vera encapsulates testbench features and assertion-based capabilities, and Synopsys promoted it as a stand-alone product (with integration into the Synopsys logic simulator). Sometime later, due to popular demand, Synopsys opened things up to for third-party use by making OpenVera™ and OpenVera Assertions (OVA) available.

Somewhere around this time, SystemVerilog was equipped with its first pass at an assert statement. Meanwhile, due to the increasing interest in formal verification technology, one of the Accellera standards committees started to look around for a formal verification language it could adopt as an industry standard. A number of languages were evaluated, including OVA, but in 2002, the committee eventually opted for the Sugar language from IBM. Just to add to the fun, Synopsys then donated OVA to the Accellera committee in charge of SystemVerilog (this was a different committee from the one evaluating formal property languages).

Yet another Accellera committee ended up in charge of something called the open verification library, or OVL, which refers to a library of assertion/property models available in both VHDL and Verilog 2K1.

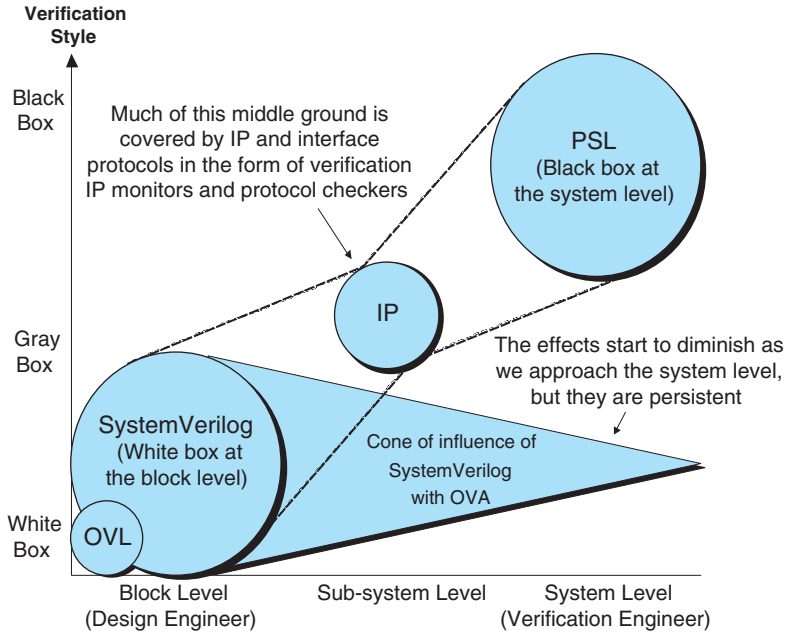

Verification Style
Black Box
Much of this middle ground is covered by IP and interface protocols in the form of verification IP monitors and protocol checkers
PSL (Black box at the system level)
IP
Gray Box
The effects start to diminish as we approach the system level, but they are persistent
SystemVerilog (White box at the block level)
Cone of influence of SystemVerilog with OVA
White Box
OVL
Block Level (Design Engineer)
Sub-system Level
System Level (Verification Engineer)

FIGURE 7-14 Trying to put everything into context and perspective.

So now we have the assert statements in VHDL and SystemVerilog, OVL (the library of models), OVA (the assertion language), and the *property specification language* (PSL), which is the Accellera version of IBM's Sugar language (Figure 7-14). The advantage of PSL is that it has a life of its own in that it can be used independently of the languages used to represent the functionality of the design itself. The disadvantage is that it doesn't look like anything the hardware description languages design engineers are familiar with, such as VHDL, Verilog, C/C++, and the like. There is some talk of spawning various flavors of PSL, such as a VHDL PSL, a Verilog PSL, a SystemC PSL, and so forth; the syntax would differ among these flavors so as to match the target language, but their semantics would be identical.

Figure 7-14 attempts to show the state of things regarding the various verification styles and languages. It's important to note that this figure just reflects one view of the world, and not everyone will agree with it (some folks will consider this to be a brilliant summation of an incredibly confusing situation, while others will regard it as being a gross simplification at best and utter twaddle at worst).

Insider Info

Don't make the mistake of referring to "PSL/Sugar" as a single/combined language. There's PSL and there's Sugar and they're not the same thing. PSL is the Accellera standard, while Sugar is the language used inside IBM.

MISCELLANEOUS

HDL to C Conversion

As we discussed in Chapter 6, there is an increasing push toward capturing designs at higher levels of abstraction such as C/C++. In addition to facilitating architectural exploration, high-level (behavioral and/or algorithmic) C/C++ models can simulate hundreds or thousands of times faster than can their HDL/RTL counterparts.

Having said this, many design engineers still prefer to work in their RTL comfort zone. The problem is that when you are simulating an entire SoC with an embedded processor core, memory, peripherals, and other logic all represented in RTL, you are lucky to achieve simulation speeds of more than a couple of hertz (that is, a few cycles of the main system clock for each second in real time).

To address this problem, some EDA companies are starting to offer ways to translate your "Golden RTL" models into faster-simulating alternatives that can achieve kilohertz simulation speeds. This is fast enough to allow you to run software on your hardware representation for milliseconds of real run time. In turn, this allows you to test critical foundation software, such as drivers, diagnostics, and firmware, thereby facilitating system validation and verification to occur much faster than with traditional methods.

Code Coverage

In the not-so-distant past, code coverage tools were specialist items provided by third-party EDA vendors. However, this capability is now considered important enough that all of the big boys have code coverage integrated into their verification (simulation) environments, but, of course, the feature sets vary among offerings.

By now, it may not surprise you to learn that there are many different flavors of code coverage, summarized briefly in order of increasing sophistication as follows:

- *Basic code coverage*: This is just line coverage; that is, how many times each line in the source code is hit (executed).
- *Branch coverage*: This refers to conditional statements like if-then-else; how many times do you go down the then path and how many down the else path.

- *Condition coverage*: This refers to statements along the lines of "if (*a* OR *b* == TRUE) then." In this case, we are interested in the number of times the then path was taken because variable *a* was TRUE compared to the number of times variable *b* was TRUE.
- *Expression coverage*: This refers to expressions like "*a* = (*b* AND *c*) OR !d". In this case, we are interested in analyzing the expression to determine all of the possible combinations of input values and also which combinations triggered a change in the output and which variables were never tested.
- *State coverage*: This refers to analyzing state machines to determine which states were visited and which ones were neglected, as well as which guard conditions and paths between states are taken, and which aren't, and so forth. You can derive this sort of information from line coverage, but you have to read between the lines (pun intended).
- *Functional coverage*: This refers to analyzing which transaction-level events (e.g., memory-read and memory-write transactions) and which specific combinations and permutations of these events have been exercised.
- *Assertion/property coverage*: This refers to a verification environment that can gather, organize, and make available for analysis the results from all of the different simulation-driven, static formal, and dynamic formal assertion-/property-based verification engines. This form of coverage can actually be split into two camps: *specification-level coverage* and *implementation-level* coverage. In this context, specification-level coverage measures verification activity with respect to items in the high-level functional or macro-architecture definition. This includes the I/O behaviors of the design, the types of transactions that can be processed (including the relationships of different transaction types to each other), and the data transformations that must occur. By comparison, implementation-level coverage measures verification activity with respect to microarchitectural details of the actual implementation. This refers to design decisions that are embedded in the RTL that result in implementation-specific corner cases, for example, the depth of a FIFO buffer and the corner cases for its "high-water mark" and "full" conditions. Such implementation details are rarely visible at the specification level.

Performance Analysis

One final feature that's important in a modern verification environment is its ability to do *performance analysis*. This refers to having some way of analyzing and reporting exactly where the simulator is spending its time. This allows you to focus on high-activity areas of your design, which may reap huge rewards in terms of final system performance.

INSTANT SUMMARY

Table 7-1 shows the major types of design tools covered in this chapter, along with their important features.

TABLE 7-1

Simulation	Event-driven logic simulators Mixed-language simulation Delay modeling Cycle-based simulators
Synthesis	HDL synthesis technology Physically aware synthesis technology Retiming Replication Resynthesis
Timing Analysis	Static timing analysis Statistical static timing analysis Dynamic timing analysis
Verification	Verification IP Bus functional models Hardware verification languages Formal verification Static formal/Dynamic formal Verification environments/languages

Chapter 8

Choosing the Right Device

In an Instant

Choosing
Technology
Basic Resources and Packaging
General-purpose I/O Interfaces
Embedded Multipliers, RAMs, etc.
Embedded Processor Cores
Gigabit I/O Capabilities
IP Availability
Speed Grades
Future FPGA Developments
Instant Summary

Definitions

Most of the terms used in this chapter you will have seen before, but here are a few definitions of some terms that may be unfamiliar:

- *Application Specific Modular Block* (ASMBL) is a new FPGA architecture that was developed by Xilinx. This is a highly modular, column-based architecture that makes use of flip-chip technology and eliminates geometric layout constraints associated with traditional chip design.
- *Field programmable analog arrays* (FPAAs) refers to ICs that can be programmed to implement analog circuits by use of flexible analog blocks and interconnect.
- *Structured ASICs* are a relatively new item and the term can mean different things depending on which vendor you're talking to. The term generally refers to there being predefined metal layers (reducing manufacturing time) and precharacterization of what is on the silicon (reducing design cycle time). Structured ASICs bridge the gap between field-programmable gate arrays and "standard-cell" ASICs.

CHOOSING

Choosing an FPGA can be a complex process because there are so many product families from the different vendors. Product lines and families from the

same vendor overlap; product lines and families from different vendors both overlap and, at the same time, sport different features and capabilities; and things are constantly changing, seemingly on a daily basis.

Before we start, it's worth noting that size isn't everything in the FPGA design world. You really need to base your FPGA selection on your design needs, such as number of I/O pins, available logic resources, availability of special functional blocks, and so forth.

Another consideration is whether you already have dealings with a certain FPGA vendor and product family, or whether you are plunging into an FPGA design for the very first time. If you already have a history with a vendor and are familiar with using its components, tools, and design flows, then you will typically stay within that vendor's offerings unless there's an overriding reason for change.

For the purposes of the remainder of these discussions, however, we'll assume that we are starting from ground zero and have no particular affiliation with any vendor. In this case, choosing the optimum device for a particular design is a daunting task.

Becoming familiar with the architectures, resources, and capabilities associated with the various product families from the different FPGA vendors demands a considerable amount of time and effort. In the real world, time-to-market pressures are so intense that design engineers typically have sufficient time to make only high-level evaluations before settling on a particular vendor, family, and device. In this case, the selected FPGA is almost certainly not the optimum component for the design, but this is the way of the world.

Given a choice, it would be wonderful to have access to some sort of FPGA selection wizard application (preferably Web based). This would allow you to choose a particular vendor, a selection of vendors, or make the search open to all vendors.

For the purposes of a basic design, the wizard should prompt you to enter estimates for such things as ASIC equivalent gates or FPGA system gates (assuming there are good definitions as to what equivalent gates and system gates are—see also Chapter 2). The wizard should also prompt for details on I/O pin requirements, I/O interface technologies, acceptable packaging options, and so forth.

In the case of a more advanced design, the wizard should prompt you for any specialist options such as gigabit transceivers or embedded functions like multipliers, adders, MACs, RAMs (both distributed and block RAM), and so forth. The wizard should also allow you to specify if you need access to embedded processor cores (hard or soft) along with selections of associated peripherals.

Last but not least, it would be nice if the wizard would prompt you as to any IP requirements (hey, since we're dreaming, let's dream on a grand scale). Finally, clicking the "Go" button would generate a report detailing the leading contenders and their capabilities (and costs).

Returning to the real world with a sickening thump, we remember that no such utility actually exists at this time, so we have to perform all of these evaluations by hand, but wouldn't it be nice…

TECHNOLOGY

One of your first choices is going to be deciding on the underlying FPGA technology. Your main options are as follows:

- *SRAM-based*: Although very flexible, this requires an external configuration device and can take up to a few seconds to be configured when the system is first powered up. Early versions of these devices could have substantial power supply requirements due to high transient startup currents, but this problem has been addressed in the current generation of devices. One key advantage of this option is that it is based on standard CMOS technology and doesn't require any esoteric process steps. This means that SRAM-based FPGAs are at the forefront of the components available with the most current technology node.
- *Antifuse-based*: Considered by many to offer the most security with regard to design IP, this also provides advantages like low power consumption, instant-on availability, and no requirement for any external configuration devices (which saves circuit board cost, space, and weight). Antifuse-based devices are also more radiation hardened than any of the other technologies, which makes them of particular interest for aerospace-type applications. On the downside, this technology is a pain to prototype with because it's OTP. Antifuse devices are also typically one or more generations behind the most current technology node because they require additional process steps compared to standard CMOS components.
- *FLASH-based*: Although considered to be more secure than SRAM-based devices, these are slightly less secure than antifuse components with regard to design IP. FLASH-based FPGAs don't require any external configuration devices, but they can be reconfigured while resident in the system if required. In the same way as antifuse components, FLASH-based devices provide advantages like instant-on capability, but are also typically one or more generations behind the most current technology node because they require additional process steps compared to standard CMOS components. Also, these devices typically offer a much smaller logic (system) gate-count than their SRAM-based counterparts.

BASIC RESOURCES AND PACKAGING

Once you've decided on the underlying technology, you need to determine which devices will satisfy your basic resource and packaging requirements. In the case of core resources, most designs are pin limited, and it's typically

only in the case of designs featuring sophisticated algorithmic processing like color space conversion that you will find yourself logic limited. Regardless of the type of design, you will need to decide on the number of I/O pins you are going to require and the approximate number of fundamental logical entities (LUTs and registers).

As discussed in Chapter 2, the combination of a LUT, register, and associated logic is called a *logic element* (LE) by some and *logic cell* (LC) by others. It is typically more useful to think in these terms as opposed to higher-level structures like slices and configurable logic blocks (CLBs) or logic array blocks (LABs) because the definition of these more sophisticated structures can vary between device families.

Next, you need to determine which components contain a sufficient number of clock domains and associated PLLs, DLLs, and digital clock managers (DCMs).

Last, but not least, if you have any particular packaging requirements in mind, it would be a really good idea to ensure that the FPGA family that has caught your eye is actually available in your desired package. (I know this seems obvious, but would you care to place a bet that no one ever slipped up on this point before?)

GENERAL-PURPOSE I/O INTERFACES

The next point to ponder is which components have configurable general-purpose I/O blocks that support the signaling standard(s) and termination technologies required to interface with the other components on the circuit board.

Let's assume that way back at the beginning of the design process, the system architects selected one or more I/O standards for use on the circuit board. Ideally, you will find an FPGA that supports this standard and also provides all of the other capabilities you require. If not, you have several options:

- If your original FPGA selection doesn't provide any must-have capabilities or functionality, you may decide to opt for another family of FPGAs (possibly from another vendor).
- If your original FPGA selection does provide some must-have capabilities or functionality, you may decide to use some external bridging devices (this is expensive and consumes board real estate). Alternatively, in conjunction with the rest of the system team, you may decide to change the circuit board architecture (this can be really expensive if the system design has progressed to any significant level).

EMBEDDED MULTIPLIERS, RAMS, ETC.

At some stage you will need to estimate the amount of distributed RAM and the number of embedded block RAMs you are going to require (along with the required widths and depths of the block RAMs).

Similarly, you will need to muse over the number of special embedded functions (and their widths and capabilities) like multipliers and adders. In the case of DSP-centric designs, some FPGAs may contain embedded functions like MACs that will be particularly useful for this class of design problem and may help to steer your component selection decisions.

EMBEDDED PROCESSOR CORES

If you wish to use an embedded processor core in your design, you will need to decide whether a soft core will suffice (such a core may be implemented across a number of device families) or if a hard core is the order of the day.

In the case of a soft core, you may decide to use the offering supplied by an FPGA vendor. In this case, you are going to become locked into using that vendor, so you need to evaluate the various alternatives carefully before taking the plunge. Alternatively, you may decide to use a third-party soft-core solution that can be implemented using devices from multiple vendors.

If you decide on a hard core, you have little option but to become locked into a particular vendor. One consideration that may affect your decision process is your existing experience with different types of processors. Let's say that you hold a black belt in designing systems based around the PowerPC, for example. In such a case, you would want to preserve your investment in PowerPC design tools and flows (and your experience and knowledge in using such tools and flows). Thus, you would probably decide on an FPGA offering from Xilinx because they support the PowerPC. Alternatively, if you are a guru with respect to ARM or MIPS processors, then selecting devices from Altera or QuickLogic, respectively, may be the way to go.

GIGABIT I/O CAPABILITIES

If your system requires the use of gigabit transceivers, then points to consider are the number of such transceivers in the device and the particular standard that's been selected by your system architects at the circuit board level.

IP AVAILABILITY

Each of the FPGA vendors has an IP portfolio. In many cases there will be significant overlap between vendors, but more esoteric functions may only be available from selected vendors, which may have an impact on your component selection.

Alternatively, you may decide to purchase your IP from a third-party provider. In such a case, this IP may be available for use with multiple FPGAs from different vendors (and a subset of device families from those vendors).

Key Concept

We commonly think of IP in terms of hardware design functions, but some IP may come in the form of software routines. For example, consider a communications function that might be realized as a hardware implementation in the FPGA fabric or as a software stack running on the embedded processor. In the latter case, you might decide to purchase the software stack routines from a third party, in which case you are essentially acquiring software IP.

SPEED GRADES

Once you've decided on a particular FPGA component for your design, one final decision is the speed grade of this device. The FPGA vendors' traditional pricing model makes the performance (speed grade) of a device a major factor with regard to the cost of that device.

—Technology Trade-offs—

As a rule of thumb, moving up a speed grade will increase performance by 12 to 15 percent, but the cost of the device will increase by 20 to 30 percent. Conversely, if you can manipulate the architecture of your design to improve performance by 12 to 15 percent (say, by adding additional pipelining stages), then you can drop a speed grade and save 20 to 30 percent on the cost of your silicon (FPGA).

If you are only contemplating a single device for prototyping applications, then this may not be a particularly significant factor for you. On the other hand, if you are going to be purchasing hundreds or thousands of these little rascals, then you should start thinking very seriously about using the lowest speed grade you can get away with.

The problem is that modifying and reverifying RTL to perform a series of what-if evaluations of alternative implementations is difficult and time-consuming. (Such evaluations may include performing certain operations in parallel versus sequentially, pipelining portions of the design versus nonpipelining, resource sharing, etc.) This means that the design team may be limited to the number of evaluations it can perform, which can result in a less-than-optimal implementation.

As discussed in Chapter 6, one alternative is to use a pure untimed C/C++-based flow. Such a flow should feature a C/C++ analysis and synthesis engine that allows you to perform microarchitecture trade-offs and evaluate their effects in terms of size/are and speed/clock cycles. Such a flow facilitates improving the performance of a design, thereby allowing it to make use of a slower speed grade if required.

Insider Info

On the bright side, once a design team has selected an FPGA vendor and become familiar with a product family, it tends to stick with that family for quite some time, which makes life (in the form of the device selection process) much easier for subsequent projects.

FUTURE FPGA DEVELOPMENTS

One thing is certain—any predictions of the future that we might make are probably going to be of interest only for the purposes of saying, "Well, we didn't see that coming, did we?" If you had told me back in 1980 when I started my career in electronics that one day we'd be designing with devices containing hundreds of millions of logic gates and the devices would be reconfigurable like today's SRAM-based FPGAs, I'd have laughed my socks off. Xilinx now has a family of 65-nm FPGA products on the market with over a billion transistors on one chip.

Super-fast I/O

When it comes to the gigabit transceivers discussed in Chapter 2, today's high-end FPGA chips typically sport one or more of these transceiver blocks, each of which has multiple channels. Each channel can carry 2.5 Gbps of real data; so four channels have to be combined to achieve 10 Gbps. Furthermore, an external device has to be employed to convert an incoming optical signal into the four channels of electrical data that are passed to the FPGA. Conversely, this device will accept four channels of electrical data from the FPGA and convert them into a single outgoing optical signal. Some FPGAs today can accept and generate these 10 Gbps optical signals internally.

Insider Info

Another technology that may come our way at some stage in the future is FPGA-to-FPGA and FPGA-to-ASIC wireless or wireless-like interchip communications. With regard to my use of the term wireless-like, I'm referring to techniques such as the experimental work currently being performed by Sun Microsystems on interchip communication based on extremely fast, low-powered capacitive coupling. This requires the affected chips to be mounted very (VERY) close to each other on the circuit board, but should offer interchip signal speeds 60 times higher than the fastest board-level interconnect technologies available today.

Super-fast Configuration

The vast majority of today's FPGAs are configured using a serial bit-stream or a parallel stream only 8 bits wide. This severely limits the way in which

these devices can be used in reconfigurable computing-type applications. Quite some time ago (somewhere around the mid-1990s), a team at Pilkington Microelectronics (PMEL) in the United Kingdom came up with a novel FPGA architecture in which the device's primary I/O pins were also used to load the configuration data. This provided a superwide bus (256 or more pins/bits) that could program the device in a jiffy.

As an example of where this sort of architecture might be applicable, consider the fact that there is a wide variety of compressor/decompressor (CODEC) algorithms that can be used to compress and decompress audio and video data. If you have a system that needs to decompress different files that were compressed using different algorithms, then you are going to need to support a variety of different CODECs.

Assuming that you wished to perform this decompression in hardware using an FPGA, then with traditional devices you would either have to implement each CODEC in its own device or as a separate area in a larger device. You wouldn't wish to reprogram the FPGA to perform the different algorithms on the fly because this would take from 1 to 2.5 seconds with a large component, which is too long for an end user to wait (we demand instant gratification these days). By comparison, in the case of the PMEL architecture, the reconfiguration data could be appended to the front of the file to be processed (Figure 8-1).

The idea was that the configuration data would flood through the wide bus, program the device in a fraction of a second, and be immediately followed by the main audio or video data file to be decompressed. If the next file to be processed required a different CODEC, then the appropriate configuration file could be used to reprogram the device.

This concept was applicable to a wide variety of applications. Unfortunately, the original incarnation of this technology fell by the wayside, but it's not

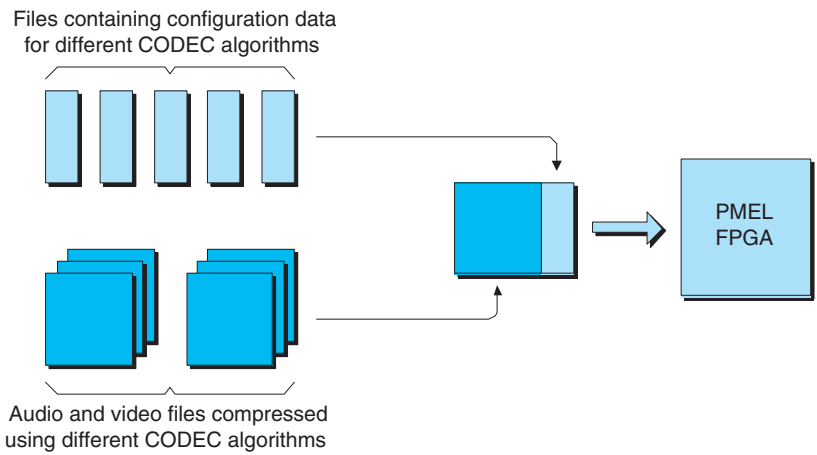


FIGURE 8-1 A wide configuration bus.

beyond the bounds of possibility that something like this could reappear in the not-so-distant future.

More Hard IP

In the case of technology nodes of 90 nm and below, it's possible to squeeze so many transistors onto a chip that we are almost certainly going to see an increased amount of hard IP blocks for such things as communications functions, special-purpose processing functions, microprocessor peripherals, and the like.

Analog and Mixed-signal Devices

Traditional digital FPGA vendors have a burning desire to grab as many of the functions on a circuit board as possible and to suck these functions into their devices. In the short term, this might mean that FPGAs start to include hard IP blocks with analog content such as analog-to-digital (A/D) and digital-to-analog (D/A) converters. Such blocks would be programmable with regard to such things as the number of quanta (width) and the dynamic range of the analog signals they support. They might also include amplification and some filtering and signal conditioning functions.

Furthermore, over the years a number of companies have promoted different flavors of field-programmable analog arrays (FPAAs). Thus, there is more than a chance that predominantly digital FPGAs will start to include areas of truly programmable analog functionality similar to that provided in pure FPAA devices.

ASMBL and Other Architectures

In 2003, Xilinx announced their Application Specific Modular BLock (ASMBL™) architecture. The idea here is that you have an underlying column-based architecture, where the folks at Xilinx have put a lot of effort into designing different flavors of columns for such things as:

General-purpose programmable logic
Memory
DSP-centric functions
Processing functions
High-speed I/O functions
Hard IP functions
Mixed-signal functions

Xilinx provides a selection of off-the-shelf devices, each with different mixes of column types targeted toward different application domains (Figure 8-2).

Different Granularity

As we discussed in Chapter 2, FPGA vendors and university students have spent a lot of time researching the relative merits of 3-, 4-, 5-, and even 6-input

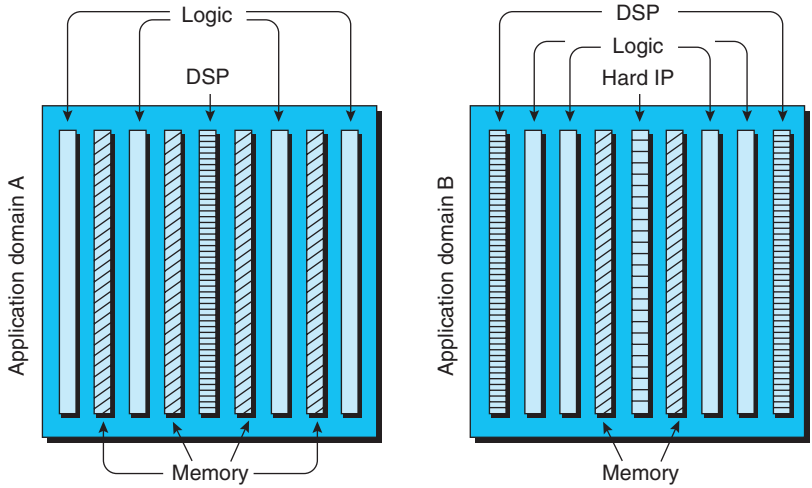


FIGURE 8-2 Using the underlying ASMBL architecture to create a variety of off-the-shelf devices with domain-specific functionality.

LUTs. In the past, some devices were created using a mixture of different LUT sizes, such as 3-input and 4-input LUTs, because this offered the promise of optimal device utilization. For a variety of reasons, the vast majority of today's FPGAs contain only 4-input LUTs, but it's not beyond the range of possibility that future offerings will sport a mixture of different LUT sizes.

Embedding FPGA Cores in ASIC Fabric

The cost of developing a modern ASIC at the 90-nm technology node is horrendous. This problem is compounded by the fact that, once you've completed a design and built the chip, your algorithms and functions are effectively "frozen in silicon." This means that if you have to make any changes in the future, you're going to have to regenerate the design, create a new set of photo-masks (costing around $1 million), and build a completely new chip.

To address these issues, some users are interested in creating ASICs with FPGA cores embedded into the fabric. Apart from anything else, this means that you can use the same design for multiple end applications without having to create new mask sets.

I also think that we are going to see increased deployment of structured ASICs and that these will lend themselves to sporting embedded FPGA cores because their design styles and tools will exhibit a lot of commonality.

MRAM-based Devices

In Chapter 1, we introduced the concept of MRAM. MRAM cells have the potential to combine the high speed of SRAM, the storage capacity of DRAM,

and the nonvolatility of FLASH, all while consuming a miniscule amount of power. MRAM-based memory chips are now available, and other devices, such as MRAM-based FPGAs, will probably start to appear soon.

Don't Forget the Design Tools

As we discussed above, some FPGAs now contain 1 billion transistors or more. Existing HDL-based design flows in which designs are captured at the RTL-level of abstraction are already starting to falter with the current generation of devices, and it won't be long before they essentially grind to a halt.

One useful step up the ladder will be increasing the level of design abstraction by using the pure C/C++-based flows introduced in Chapter 6. Really required, however, are true system-level design environments that help users explore the design space at an extremely high level of abstraction. In addition to algorithmic modeling and verification, these environments will aid in partitioning the design into its hardware and software components.

These system-level environments will also need to provide performance analysis capabilities to aid users in evaluating which blocks are too slow when realized in software and, thus, need to be implemented in hardware, and which blocks realized in hardware should really be implemented in software so as to optimize the use of the chip's resources.

People have been talking about this sort of thing for ages, and various available environments and tools go some way toward addressing these issues. In reality, however, such applications have a long way to go with regard to their capabilities and ease of use.

Expect the Unexpected

Before closing, I'd just like to reiterate that anything you or I might guess at for the future is likely to be a shallow reflection of what actually comes to pass. There are device technologies and design tools that have yet to be conceived, and when they eventually appear on the stage (and based on past experience, this will be sooner than we think), we are all going to say, "WOW! What a cool idea!" and "Why didn't I think of that?" Good grief, I LOVE electronics!

INSTANT SUMMARY

Table 8-1 shows the general approach for choosing an FPGA device.

TABLE 8-1 Choosing an FPGA

1.	Already deal with a specific vendor or product family? Any compelling reasons to change?
2.	Which technology to use?
3.	ASIC equivalent gates or FPGA system gates?
4.	I/O pin requirements?
5.	I/O interface technology?
6.	Acceptable packaging options?
7.	Need special options such as gigabit transceivers or embedded functions like adders, multipliers, MACs, RAMS?
8.	Need embedded processor cores? If so, hard or soft?
9.	IP requirements?
10.	Which speed grade?

Index

A

16-bit shift register, 53
AND gate, 6
AND gate, 6
Altera, 24, 25, 43, 68, 142, 144, 189
Amorphous silicon, 7, 8
Analog and mixed-signal devices, 193
Antifuse-based FPGA, 16–17, 51, 187
Antifuse technology, 7–8
Application programming interface (API), 96
Application-specific integrated circuits (ASICs), 2
Application-specific standard parts (ASSPs), 2
Applications, of FPGAs, 3–4
Architectural features, of FPGAs:
  CLBs vs. LABs vs. slices, 23
    CLBs and LABs, 25–6
    distributed RAMs, 26
    logic cell (LC), 24
    logic element (LE), 24
    shift registers, 26–7
    slicing and dicing, 24–5
  clock managers, 33–6
  clock trees, 32–3
  coarse-grained architecture, 18
  core voltages vs. I/O supply voltages, 37–8
  embedded adders, 28–9
  embedded multipliers, 27–9
  embedded processor cores, 29
    hard microprocessor cores, 30–1
    soft microprocessor cores, 31–2
  embedded RAMs, 27
  fine-grained architecture, 18, 19
  general-purpose I/O:
    configurable I/O impedances, 37
    configurable I/O standards, 36–7
  gigabit transceivers, 38–40
    multiple standards, 39
  intellectual property (IP), 40–4
    block/core generators, 43–4
    handcrafted IP, 41–3
  logic blocks:
    LUT-based approach, 20–1
    LUT vs. distributed RAM vs. SR, 22–3
    MUX-based approach, 19–20
  medium-grained architecture, 19
  programming technologies:
    antifuse-based devices, 16–17
    E^2PROM/FLASH-based devices, 17–18
    hybrid FLASH-SRAM devices, 18
    SRAM-based devices, 14–15
  system gates vs. real gates, 44–6
Architecturally aware FPGA flows, 93
ASIC fabric, embedded FPGA cores in, 194
ASIC hardware, 126
ASIC-to-FPGA design, 71–2
ASIC vs. FPGA designs, 61
  asynchronous design practices:
    asynchronous structures, 65
    combinational loops, 65
    delay chains, 65
  clock considerations:
    clock balancing, 65–6
    clock domains, 65
    clock gating vs. clock enabling, 66
    PLLs and clock conditioning circuitry, 66
    reliable data transfer, across multiclock domains, 66
  coding styles, 62
  flip-flops, with set and reset inputs, 67
  global resets and initial conditions, 67
  latches, 67
  levels of logic, 64
  migration:
    ASIC-to-FPGA, 71–2
    FPGA only, 69
    FPGA-to-ASIC, 70–1
    FPGA-to-FPGA, 69–70
  pipelining, 62–3
  resource sharing, 67–8
  state machine encoding, 68
  test methodologies, 69
ASMBL architecture, 185, 193
Assertion, definition of, 176
Assertion-based verification (ABV), 175

Assertions/properties, 175, 176, 183
specification techniques, 178–9
Asynchronous design practices:
asynchronous structures, 65
combinational loops, 65
delay chains, 65
Augmented C/C++-based flows, 117
alternatives, 119–20
Automated reasoning, 175
Automatic test pattern generation (ATPG), 69
Auto-skew correction, in clock managers, 35–6

B

Back-end tools, 81
Basic code coverage, 182
Basic resources and packaging requirements, 187–8
Behavioral level of abstraction, 91
Bidirectional buffers, 106
Bit files, 49
Bitstream encryption, 15
Block-level IP environments, 138–9
Block RAM, 27
Boolean equations, 90, 91
Boundary scan technique, 58–9
Branch coverage, 182
Built-in self-test (BIST), 69
Bus functional models (BFMs), 172
Bus interface model (BIM), 108
Bytes, 55

C

C vs. C++, 110–11
C/C++-based design flow, 108
ASIC vs. FPGA, 70–1
C/C++ model, of CPU, 150
C/C++ synthesis abstraction, different levels of, 123–4
Cadence Design Systems, 97
Case statement synthesis, 104
Central processing unit (CPU), 140, 141, 148, 149, 150
Choosing an FPGA, 185
basic resources and packaging, 187–8
embedded multipliers and RAMS, 188–9
embedded processor cores, 189
future FPGA developments, 191
analog and mixed-signal devices, 193
ASIC fabric, embedded FPGA cores in, 194
ASMBL, 193
design tools, 195
different granularity, 193–4
hard IP blocks, 193
MRAM-based devices, 194–5
super-fast configuration, 191–3
super-fast I/O, 191
general-purpose I/O interfaces, 188
gigabit I/O capabilities, 189
IP availability, 189
speed grades, 190
technology, 187
CLBs vs. LABs vs. Slices, 23
CLBs and LABs, 25–6
distributed RAMs, 26
logic cell (LC), 24
logic element (LE), 24
shift registers, 26–7
slicing and dicing, 24–5
Clock balancing, 65–6
Clock domains, 65, 71
Clock gating vs. clock enabling, 66
Clock managers:
auto-skew correction, 35–6
frequency synthesis, 33–4
jitter removal, 33, 34
phase shifting, 34–5
Clock trees, 32–3
Clunky flat schematics, 86, 87
Coarse-grained architecture, 18
Code coverage, 182–3
Co-Design Automation, 101
Combinational loops, 61, 65
Compressor/decompressor (CODEC) algorithm, 192
Computer-aided design (CAD), 81
Computer-aided engineering (CAE), 81
Condition coverage, 183
Configurable I/O impedances, 37
Configurable I/O standards, 36–7
Configurable logic block (CLB), 25–6, 83, 85
Configuration bitstream, 49
Configuration cells, 50–1
Configuration commands, 49
Configuration data, 49
Configuration files, 49
Configuration port usage, 53
parallel load:
with FPGA as master, 55–6
with FPGA as slave, 56–7
serial load:
with FPGA as master, 54–5
with FPGA as slave, 57–8
Constants usage, 104

Constraints, 177
Core voltages vs. supply voltage, 37–8
Corner case, 180
Corner condition, 179–80
CoWare, 3, 127
Cycle-based simulators, 163–5

D

D-type flip-flop, 79, 80
Daisy-chaining FPGAs, 55
Daughter clocks, 33, 34, 35
Declarative, 177
Deep submicron (DSM), 14
Delay chains, 61, 65
Delay-locked loops (DLLs), 36
Design Automation Conference (DAC), 99
Design flows, 108
  augmented C/C++-based flows, 117
    alternatives, 119–20
  C vs. C++, 110–11
  C/C++ model, of CPU, 150
  C/C++ synthesis abstraction, different levels of, 123–4
  C/C++-based design flows, 108
  concurrent vs. sequential nature, of C/C++, 110–11
  DSP-based design flows, 125–6
  DSP implementations, 126
    DSP algorithm, 128–30
    DSP-related embedded FPGA resources, 130–1
    software running on DSP core, 127–8
    system-level evaluation and algorithmic verification, 126–7
  embedded processor-based design flows, 140
    hard microprocessor cores, 142–3
    soft microprocessor cores, 143–4
  FPGA, using:
    as its own developmental environment, 147
  FPGA-centric design flows, for DSPs, 131
    block-level IP environments, 138–9
    domain-specific languages, 131–3
    floating-point vs. fixed-point representations, 133–4
    system-level design and simulation environments, 133
    system/algorithmic level, to C/C++, 137–8
    system/algorithmic level, to RTL, 134–6, 136–7
    testbench, 139
    hardware modeler, physical chip in, 150
    instruction set simulator, 151
    mixed DSP and VHDL/Verilog environments, 139–40
    mixed-language design and verification environments, 124–5
    partitioning design, into hardware and software components, 145–7
    pure C/C++-based flows, 120–3
    RTL model, of CPU, 150
    systemC-based flows, 112
      alternatives, 114–17
      levels of abstraction, 113–14
    visibility improvement, in design, 147–8
Design tools, 195
  code coverage, 182–3
  formal verification, 174
    assertion/property specification techniques, 178–9
    flavors, 174–6
    static formal vs. dynamic formal verification, 179–82
    terminologies, 176–7
  HDL to C conversion, 182
  logic/HDL synthesis technology, 166
  performance analysis, 183
  physically aware synthesis technology, 167
  replication, 168–9
  resynthesis, 169
  retiming, 168
  simulation tool:
    cycle-based simulators, 163–165
    delay modeling, 160–3
    event-driven logic simulators, 156–8
    logic simulator, choosing, 165–6
    logic values, 158–9
    mixed-language simulation, 159–60
  timing analysis:
    static timing analysis (STA), 169–70
    statistical static timing analysis, 170–1
  verification:
    environments, 173
    simulation results, analyzing, 174
    testbenches creation, 173
    verification IP, 171–2
Device under test (DUT), 171, 172
Digital signal processing (DSP), 3–4
  DSP-based design flows, 125
    alternative implementations, 126–31
    FPGA-centric design flows, 131–9
    mixed DSP and VHDL/Verilog environments, 139–40
Direct Programming Interface (DPI), 102

Distributed RAMs, 26, 53, 188
Domain-specific languages (DSLs), 108, 131–2
DSP algorithm, 128–30
DSP implementations, 126
  DSP algorithms, 128–30
  DSP-related embedded FPGA resources, 130–1
  software running, on DSP core, 127–8
  system-level evaluation and algorithmic verification, 126–7
DSP-based design flows, 125–6
DSP-related embedded FPGA resources, 130–1
Dynamic formal verification, 175
  vs. static formal verification, 179–82
Dynamic RAM (DRAM), 8

E

e verification language, 173
e-RAMs, 27
E^2PROM-based devices, 17–18
EDA vendors, 70, 88, 89, 94, 103
Electrically erasable PLDs (EEPLDs/E^2PLDs), 11
Electrically erasable programmable read-only memories (EEPROMs/E^2PROMs), 11
Electronic design automation (EDA), 75, 81
Electronic system level (ESL) environment, 145
Embedded adders, 28–9
Embedded microcontrollers, 4
Embedded multipliers, 27–9, 189
Embedded processor-based design flows, 140
  hard microprocessor cores, 142–3
  soft microprocessor cores, 143–4
Embedded processor cores, 59–60, 29, 189
  hard microprocessor cores, 30–1
  soft microprocessor cores, 31–2
Embedded RAMs, 27
Emitter-coupled logic (ECL), 162, 28
EPROM, *see* Erasable programmable read-only memory
Erasable PLDs (EPLDs), 11
Erasable programmable read-only memory (EPROM), 9, 10, 11
Event wheel, 156–7
Event, 177
Event-driven logic simulators, 156–8
Expression coverage, 183

F

Fast carry chain, 27
Fast Fourier transform (FFT), 96
Field-effect transistor (FET), 13–14
Field programmable analog arrays (FPAAs), 185, 193
Field programmable gate arrays (FPGAs)
  applications, 3–4
  definition, 1
  need for, 1–3
  technologies:
    antifuse technology, 7–8
    FLASH-based technologies, 9–11
    fusible-link technology, 4–6
    SRAM-based technology, 8–9
Fine-grained architecture, 18, 19
Firm IP, 41
Fixed-point representations:
  vs. floating-point representations, 133–4
FLASH-based devices, 17–18, 51
FLASH-based FPGA, 17, 187
FLASH-based technologies, 9–11
Flat vs. hierarchical schematics:
  clunky flat schematics, 86, 87
  hierarchical block-based schematics, 87–8
Flip-flops, with set and reset inputs, 67
Floating-point representations:
  vs. fixed-point representations, 133–4
Foreign language translation, 159
Formal verification (FV), 174
  assertion/property specification techniques, 178–9
  flavors, 174–6
  static formal vs. dynamic formal verification, 179–82
  terminologies, 176–7
FPGA-centric design flows, for DSPs, 131
  block-level IP environments, 138–9
  domain-specific languages, 131–3
  floating-point vs. fixed-point representations, 133–4
  system-level design and simulation environments, 133
  system/algorithmic level:
    to C/C++, 137–8
    to RTL, 134–6, 136–7
  testbench, 139
FPGA only design, 69
FPGA-to-ASIC design, 70–1
FPGA-to-FPGA design, 69–70
FPGA vendors, 14, 24, 40, 43, 45, 70, 88, 103, 144, 166, 186, 189, 190
"fred-in-the-shed"-type operations, 2
"frozen in silicon", 2, 194
Functional coverage, 183
Functional level of abstraction, 90
Functional verification, 76, 83, 171

Fusible-link technology, 4–6
Future FPGA developments, 191
  analog and mixed-signal devices, 193
  ASIC fabric, embedded FPGA cores in, 194
  ASMBL architecture, 193
  design tools, 195
  different granularity, 193–4
  hard IP blocks, 193
  MRAM-based devices, 194–5
  super-fast configuration, 191–3
  super-fast I/O, 191
"fuzzy" clock, 33, 34

G

Gate-level design, 75, 76
Gate-level netlist, 76, 77, 78, 79, 80, 86, 90, 92
Gate-level netlist and pure untimed C/C++, 122
Gate level of abstraction, 90
Gateway Design Automation, 96, 97
General-purpose I/O interfaces, 188
General-purpose microprocessor, 126, 140
Gigabit I/O capabilities, 189
Gigabit transceivers, 38
  multiple standards, 39–40
Global resets and initial conditions, 67
Graphical design entry, 95–6

H

Handcrafted IP, 41–3
Hard IP blocks, 40, 41, 193
Hard microprocessor core, 30–1, 108, 142–3
Hard-wired mode pins, 54
Hardware description languages (HDLs), 75
Hardware description languages (HDL)-based design flows:
  architecturally aware FPGA flows, 93
  bidirectional buffers, 106
  constants usage, 104
  graphical design entry, 95–6
  internal tri-state buffers, 105–6
  latchware inference, 104
  levels of abstraction, 89–91
  logic vs. Physically aware synthesis, 93–4
  mixed-language designs, 99–100
  resource sharing, 104–5
  serial vs. parallel multiplexers, 103–4
  simple (early) HDL-based ASIC flow, 91–2
  simple (early) HDL-based FPGA flow, 92–3
  Superlog and SystemVerilog, 101–2
  SystemC, 102
  unified design language for integrated circuits, 100–1
  Verilog HDL, 96–8
  VHDL and VITAL, 98–9
Hardware modeler, physical chip in, 150
Hardware verification languages (HVLs), 173
HDL synthesis technology, 166
HDL to C conversion, 182
HDL Wars, 100
Hierarchical block-based schematics, 87–8
HL delay, 160, 161
Hybrid FLASH-SRAM devices, 18
IEEE 1364, 98

I

Implementation-level coverage, 183
"Incontext" assertion/properties, 177
Inertial delay model, 162
Instruction set simulator (ISS), 108, 151
In-system programmable (ISP) device, 1
Integrated development environment (IDE), 144
Intellectual property (IP), 14, 40
  availability, 189
  handcrafted IP, 41–3
  IP block/core generators, 43–4

J

Jitter removal, in clock managers, 33, 34
JTAG port usage, 58–9

L

Language interference manual (LRM), 98
Latches, 67
Latchware inference, 104
Latency, 61, 64
Levels of abstraction, 89–91
Levels of logic, 61
LH delay, 160, 161
Linux, 112
Logic array block (LAB), 25
Logic blocks:
  LUT-based approach, 20–1
  LUT vs. distributed RAM vs. SR, 22–3
  MUX-based approach, 19–20
Logic cell (LC), 24, 188
Logic element (LE), 24, 188
Logic minimization, 75
Logic simulation, 156
Logic simulators, 76, 78–9
  choosing, 165–6
Logic values, 158–9
Logic vs. physically aware synthesis, 93–4

Logic/HDL synthesis technology, 166
LUT-based logic block, 20–1
LUT vs. distributed RAM vs. SR, 22–3
LUT/CLB-level netlist, 147

M

MAC, *see* Multiply-and-accumulate
Main FPGA fabric, 30–1
Main memory (MEM), 140
Mapping, 84
Mathworks, The, 126, 127, 133
MATLAB, 127, 132
Medium-grained architecture, 19
Microarchitecture definition tasks, 108
Microblaze, 144
Microcontroller, 108
Microprocessor unit (MPU), 140
Microsecond logic, 29, 145
Millisecond logic, 29–30, 145
Min:typ:max delays, 160, 161
Mixed-language designs, 99–100
  and verification environments, 124–5
Mixed-language simulation, 159–60
Mixed-level design capture environment, 95
Model checking, 174
MOS transistor, 9, 10
MRAM-based devices, 194–5
Multichip module (MCM), 30, 142
Multiple programming chain, 53
Multiply-and-accumulate (MAC), 28–9, 131
MUX-based logic block, 19–20

N

Nios, 144
Nonrecurring engineering (NRE), 2
NOT gate, 6

O

"One-hot" encoding scheme, 68
One-time programmable (OTP) device, 1, 6, 51
Open International Verilog (OVI), 98
Open verification library (OVL), 180, 181
OpenVera Assertions (OVA), 180, 181
OpenVera™, 180
Optimization, 75
OVA language, 178

P

Packing, 84
Parallel load:
  with FPGA as master, 55–6, 57
  with FPGA as slave, 56–7
Parallel multiplexers vs. serial multiplexers, 103–4
Parallel statement, in augmented C/C++ language, 118
Partitioning design, into hardware and software components, 145–7
Performance analysis, 183
Phase-locked loops (PLLs), 36
  and clock conditioning circuitry, 66
Physical layer communications, 4
Physically aware synthesis technology, 167
PicoBlaze, 144
Picosecond and nanosecond logic, 29, 145
Pilkington Microelectronics (PMEL), 192
Pipelined design, 163, 164
Pipelining, 61, 62–3
Place-and-route software, 82, 85
PLDs, *see* Programmable logic devices
Polygonal editors, 81
Post-place-and-route simulation, 86
Pragma, 107, 177, 179
Procedural, 177
Programmable logic devices (PLDs), 1–2
Programming (configuring), an FPGA, 49
  antifuse-based FPGAs, 51
  configuration cells, 50–1
  configuration port usage, 53
    parallel load with FPGA, as master, 55–6
    parallel load with FPGA, as slave, 56–7
    serial load with FPGA, as master, 54–5
    serial load with FPGA, as slave, 57–8
  embedded processor usage, 59–60
  JTAG port usage, 58–9
  SRAM-based FPGAS, 51
    distributed RAMs, 53
    multiple programming chains, 53
    programming embedded (block) RAMs, 52
    quickly reinitializing device, 53
Programming embedded (block) RAMs, 52
Programming language interface (PLI), 96
Programming technologies, of FPGAs:
  antifuse-based devices, 16–17
  E^2PROM/FLASH-based devices, 17–18
  hybrid FLASH-SRAM devices, 18
  SRAM-based devices, 14
    security issues, 15
Property, definition of, 176
Property specification language (PSL), 178, 181
Pure C/C++-based flows, 120–3

Q

Quantization, 134
Quantizers, 134
Quickly reinitializing device, 53

R

Real gates vs. system gates, 44–6
Reconfigurable computing (RC), 4
Register transfer level (RTL), 41, 75, 90, 108–9, 110, 150, 171
Reliable data transfer across multiclock domains, 66
Replication, 168–9
Resource sharing, 67–8, 104–5
Resynthesis, 169
Retiming, 168

S

SCAN chain insertion, 69
Schematic-based design flows, 76
  back-end tools, 81
  CAE and CAD tools, 81
  flat vs. hierarchical schematics, 86–8
  schematic-driven FPGA design flows, 88
  simple (early) schematic-driven ASIC flow, 81–3
  simple (early) schematic-driven FPGA flow, 83–6
Schematic-driven FPGA design flows, 88
Schematic diagrams, 75, 76, 77
Sequential statement, in augmented C/C++ language, 118
Serial load:
  with FPGA, as master, 54–5
  with FPGA, as slave, 57–8
Serial multiplexers vs. parallel multiplexers, 103–4
Shift registers, 26–7
Signal Processing Worksystem (SPW), 127
Simple (early) HDL-based ASIC flow, 91–2
Simple (early) HDL-based FPGA flow, 92–3
Simple (early) schematic-driven ASIC flow, 81–2
Simple (early) schematic-driven FPGA flow, 83
  mapping, 84
  packing, 84
  place-and-route, 85
  post-place-and-route simulation, 86
  timing analysis, 86
Simulation primitives, 77
Simulation tools:
  cycle-based simulators, 163–5
  delay modeling, 160–3
  event-driven logic simulators, 156–8
  logic simulator, choosing, 165–6
  logic values, 158–9
  mixed-language simulation, 159–60
Simulink®, 127, 133
Skew, 32, 35
Slicing, and dicing, 24–5
Soft core, 108
Soft IP, 40–1
Soft microprocessor cores , 31–2, 143–4
Special languages, 178
Specification-level coverage, 183
Speed grades, 190
SRAM-based devices, 14–15
SRAM-based FPGAS, 51, 187
  multiple programming chains, 53
  programming embedded (Block) RAMs, 52–3
  quickly reinitializing device, 53
SRAM configuration cells, visualization of, 52
Standard delay format (SDF), 86, 97, 158
State coverage, 183
State machine encoding, 68
State variables, 179
Static formal verification, 175
  vs. dynamic formal verification, 179–82
Static RAM (SRAM)-based technology, 8–9
Static timing analysis (STA), 86, 160, 169–70
Statistical static timing analysis, 170–1
Stimulus, 78–9
  see also Test vectors
Structured ASICs, 185
Sugar language, 178, 182
Super-fast configuration, of FPGA, 191–3
Super-fast I/O, of FPGA, 191
Superlog, and SystemVerilog, 101–2
Supply voltage vs. core voltages, 37–8
Switch level of abstraction, 90
Synopsis, 101
Synthesis:
  logic/HDL synthesis technology, 166
  physically aware synthesis technology, 167
  replication, 168–9
  resynthesis, 169
  retiming, 168
Synthesis tools, 93, 94
System/algorithmic level:
  to C/C++, 137–8
  to RTL, 134–6, 136–7
System gates vs. real gates, 44–6
System-level design and simulation environments, 133

System level of abstraction, 91
System-level evaluation and algorithmic verification, 126–7
System-on-chip (SoC), 3, 110
SystemC, 102, 112
SystemC 1.0, 112
SystemC 2.0, 112, 116
SystemC-based flows, 112
  alternatives, 114–17
  levels of abstraction, 113–14
SystemVerilog, 101–2, 178
SystemVerilog 3.0, 101

T

Technology background, of FPGAs, 187
  antifuse technology, 7–8
  FLASH-based technologies, 9–11
  fusible-link technology, 4–6
  SRAM-based technology, 8–9
Test vectors, 78–9
  see also Stimulus
Testbench, 139, 173
"The Stripe", 30
Three-band delay model, 163, 164
Time-division multiplexing (TDM), *see* Resource sharing
Timebase unit, 160
Timing analysis, 86
  static timing analysis, 169–70
  statistical static timing analysis, 170–1
Timing analysis program, 82
Timing verification, 76
"Traditional" design flows, 75
  HDL-based design flows, 89–106
  schematic-based design flows, 76–83
    back-end tools like layout, 81
    CAE and CAD tools, 81
    flat vs. hierarchical schematics, 86–8
    schematic-driven FPGA design flows today, 88
    simple (early) schematic-driven ASIC flow, 81–3
    simple (early) schematic-driven FPGA flow, 83–6
Transistor-transistor logic (TTL), 162
Transmission gate-based LUT, 21
Transport delay model, 162–3
Tri-state buffers, 105–6
Turing machines, 128

U

Ultradeep submicron (UDSM), 14
Unified design language for integrated circuits (UDL/I), 100–1
"Use-it-or-lose-it" considerations, 67

V

Value change dump (VCD) file, 158, 174
Vera®, 180
Verification:
  environments, 173
  languages, 180–1
  simulation results, analyzing, 174
  testbenches creation, 173
  verification IP, 171–2
Verilog, 159, 179
Verilog HDL, 96–8
  vs.VHDL, 100
Very high speed integrated circuit (VHSIC) program, 98
VHDL Initiative toward ASIC Libraries (VITAL), 99
VHDL International, 101
VHSIC HDL (VHDL), 98–9, 159
  vs. Verilog, 100
  and VITAL, 98–9
Visibility improvement, in design, 147–8
Vulcan Death Grip, 88

W

Wizard application, 186

X

Xilinx, 144, 191, 193
Xilinx LC, 24